LUCIANO CÉSAR ASCARI

TESTE BASEADO EM DEFEITOS DE CLASSES JAVA UTILIZANDO ASPECTOS E MUTAÇÃO DE ESPECIFICAÇÕES OCL

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora: Profa. Dra. Silvia Regina Vergilio

CURITIBA

2009

LUCIANO CÉSAR ASCARI

TESTE BASEADO EM DEFEITOS DE CLASSES JAVA UTILIZANDO ASPECTOS E MUTAÇÃO DE ESPECIFICAÇÕES OCL

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora: Profa. Dra. Silvia Regina Vergilio

CURITIBA

2009

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

RESUMO

Esse trabalho apresenta uma abordagem de teste de software orientado a objetos baseado em especificações OCL e Programação Orientada a Aspectos (POA) a abordagem utiliza as pré e pós-condições definidas para os métodos das classes, e possui dois usos principais: a geração de dados de teste a partir da especificação para o teste da implementação; e a validação da especificação e da implementação por meio da análise de especificações que sofreram mutação. O teste de software baseado na especificação contribui para identificar defeitos na especificação relacionados a caminhos ausentes. O teste de software utilizando a técnica baseada em defeitos e instrumentado com a POA apresenta como vantagens a não alteração da classe em teste, e um menor custo de execução. Para validar a abordagem proposta foi implementada a ferramenta **MuSA** (teste de **Mu**tação baseado em E**S**pecificações OCL e **A**spectos). A ferramenta MuSA foi utilizada em um experimento de avaliação que comprovou a aplicabilidade da abordagem e permitiu a comparação com critérios estruturais.

ABSTRACT

This work presents a testing approach for object oriented software, which is based on OCL and Aspect-Oriented Programming (AOP). It uses pre and post-conditions defined to the class methods under test. Two main uses are introduced to the approach: to generate test data from the specification to test the implemented code, and to validade the especification and the implementation by considering mutated specifications. The test data based on the specification aim at the discovering of missing paths in the program. The use of AOP to instrument the fault based testing is advantageous because the code of the classes do not need to be altered and a lower number of executions is required. To validate the proposed approach, a tool named **MuSA** (**Mu**tation testing based on OCL **S**pecifications and **A**spects) was implemented and used in an evaluation experiment. The evaluation results show the applicability of the approach and allow comparasion with structural criteria.

CAPÍTULO 1

INTRODUÇÃO

Obter qualidade de software é um dos principais objetivos da Engenharia de Software (ES). Mesmo com o aprimoramento constante dos métodos, técnicas e ferramentas de desenvolvimento de software, defeitos podem ser introduzidos, e somente uma atividade específica de teste pode descobrir a ocorrência de determinados defeitos no software.

O teste de software em sentido mais amplo pode ser visto como uma atividade de Verificação, Validação e Teste (VV&T), e tem como objetivo encontrar o maior número possível de erros com uma quantidade de esforço gerenciável aplicada durante um intervalo de tempo realístico [Pre06].

O teste de software pode ser classificado pela forma do teste [Pre06], que pode envolver a especificação ou o programa (código fonte). E o teste pode ser realizado nas diversas etapas de um projeto de software, e diferentes técnicas de teste são adequadas em cada uma das etapas. À cada técnica estão associados critérios de teste. Um critério é um predicado a ser satisfeito pelos casos de teste e pode ser utilizado para guiar a seleção de dados de entrada, bem como para avaliar um conjunto de casos de teste. O critério fornece uma medida de cobertura dada pelo número de elementos requeridos que foram satisfeitos pelos casos de teste. Essa cobertura pode ser utilizada para decidir se o programa foi testado o suficiente e encerrar a atividade de teste.

As técnicas de teste existentes são geralmente classificadas entre três grupos:

- Técnica estrutural: Conhecida como técnica de caixa branca, na qual o testador tem conhecimento da estrutura do programa. Os principais critérios estruturais são: critérios baseados em fluxo de controle [GG75], critérios baseados em complexidade [MB89] e critérios baseados em fluxo de dados [MBV+].

- Técnica funcional: Conhecida como técnica de caixa preta, trata-se de teste no qual o testador não tem conhecimento do código fonte, sendo um teste de análise de

entradas fornecidas e saídas retornadas pelo software. Particionamento em Classes de Equivalência e Análise de Valor Limite estão entre os principais critérios dessa técnica [Pre06].

- Técnica baseada em defeitos: Utiliza as informações dos tipos de defeitos mais freqüentes no processo de desenvolvimento de software para derivar os casos de teste. Os critérios que se destacam nessa técnica são: Semeadura de Erros e Análise de Mutantes [DLS78].

Os critérios baseados em defeitos têm se mostrado os mais eficazes em termos de número de defeitos revelados em diversos experimentos [SFB+07]. O critério Análise de Mutantes tem como objetivo revelar defeitos em um determinado programa ou especificação a partir da comparação dos resultados obtidos pela sua execução com um determinado conjunto de casos de teste, com os resultados obtidos pela execução de programas mutantes, que são previamente definidos. Entretanto, o teste baseado em mutação de programas é bastante caro em termos do número de execuções do programa e seus mutantes.

Outra limitação da técnica baseada em defeitos e também da técnica estrutural aplicadas aos programas é a existência de caminhos ausentes. Esses caminhos estão relacionados a certas funcionalidades que estão presentes na especificação, mas que por engano não foram implementadas. Dados de teste derivados somente considerando a implementação não garantem que esse tipo de defeito seja descoberto.

Outro ponto a ser considerado é que mudanças no paradigma no desenvolvimento de software refletem diretamente em adequações no processo de teste. O software desenvolvido no paradigma Orientado a Objetos (OO) tem características específicas. Os novos conceitos que surgiram com o paradigma OO (encapsulamento, herança, polimorfismo, etc) apresentam-se como novos desafios no que diz respeito a teste de software, pois novos tipos de defeitos precisam ser considerados. A diferença com relação ao desenvolvimento no paradigma procedimental fez com que as técnicas e critérios tradicionais de teste de software sofressem adaptações que correspondessem à necessidade desse paradigma específico.

Por exemplo, no teste de programa OO destacam-se os estudos realizados na adequação

e inovação dos critérios estruturais baseados em fluxo de controle e de dados e, dos critérios baseados em mutação, antes aplicados no paradigma procedimental [HR94].

Harrold e Rothermel [HR94] propuseram critérios baseados em fluxo de dados para o teste de classes. Considerando pares de definições e uso de variáveis que permitem avaliar relações de fluxo de dados em programas OO.

Uma ferramenta de teste de programa OO é a JaBUTi (*Java Bytecode Understanding and Testing*) [VMWD05]. Ela realiza o teste de software desenvolvido com a linguagem Java. Uma das suas características diferenciais, é que sua análise estática é executada sobre o código compilado em Java, ou seja, sobre os *bytecodes*. A ferramenta de teste JaBUTi permite a utilização de diversos critérios estruturais e implementa critérios baseados em fluxo de dados e em fluxo de controle.

O teste de software em OO também é realizado com base em modelos de especificação. Em alguns casos, os modelos são utilizados como base para a geração de casos de teste e validação da implementação. Essa forma de teste é interessante para comprovar a corretude e completude da implementação com relação ao que foi especificado.

Nesse contexto, muitos autores têm utilizado modelos baseados em contrato [VJ03] e [JHS+05]. Um contrato é parte de uma especificação de um sofware, que define obrigações entre as partes (fornecedor do serviço e cliente do serviço), assim como os resultados do seu relacionamento. Existem diversas linguagens de especificação para escrever contratos, entre elas destaca-se a OCL *(Object Constraint Language Specification)* [OMG08].

A OCL é uma linguagem de expressão que pode ser usada para especificar invariantes, pré-condições, pós-condições e outros tipos de restrições que os modelos UML não representam a contento. Apesar de possuir algumas limitações, a OCL é hoje a linguagem de expressão mais usada para especificar formalmente modelos no contexto de software orientados a objetos [AS05].

A OCL tem sido utilizada para geração de dados de teste com diferentes objetivos. Alguns trabalhos têm o objetivo de validar a especificação (ou algum modelo de especificação). A maioria dos trabalhos utilizam as especificações OCL para gerar dados de teste, com o objetivo de realizar um teste funcional ou de caixa preta.

Por exemplo, o trabalho de Benattou *et al.* [BBH02] gera dados de teste a partir de especificações OCL baseando-se no particionamento em classes de equivalência. Também considerando o particionamento em classes, Aertryck e Jensen [VJ03] apresentam uma ferramenta chamada UML-Casting para gerar dados de teste com base em diagramas de estado da UML. Desses diagramas, restrições OCL são obtidas e cobertas pelos dados de teste. Aicherming e Salas [AS05] exploram a geração de dados de teste como um problema de satisfação de uma restrição. As restrições, em formato BNF, são obtidas de uma especificação OCL e, também de especificações que sofreram mutação. Contudo, eles não introduzem um conjunto de operadores de mutação nesse contexto e nem implementam uma ferramenta.

O problema dos trabalhos mencionados acima é que eles visam ao teste da especificação e não utilizam a mesma para testar a implementação. Há na literatura, uma outra linha de trabalho que tem esse objetivo. Buscando a utilização de uma linguagem de especificação mais simples, Jiang *et al.* [JHS+05] apresentam uma proposta de mutação para teste de componentes de software. A aplicação de mutação é feita sobre os contratos definidos aos componentes em uma linguagem própria, estendida da linguagem de definição de *Enterprise Java Beans (EJB)*. O conjunto de operadores de mutação aplicado é definido pelos autores no trabalho. Utilizando os mesmos operadores de mutação de Jiang *et al.*, Mei e Zhang [MZ05] apresentam uma ferramenta para teste baseado em contrato de *Web Services* (WS).

Há trabalhos que têm como objetivo dinamicamente testar restrições OCL para checar inconsistências automaticamente, gerar código, e etc. Gogolla *et al.* [GBR03] descrevem uma ferramenta chamada USE (UML-based Specification Enviroment) para checar inconsistências em diagramas de classe com base na OCL. A ferramenta SPACES (Specification bAsed Component tESter) [BLM+07] gera dados de teste para componentes em Java a partir de diagramas em UML e restrições OCL.

Mais recentemente, o teste de software orientado a objetos tem se beneficiado com a utilização da Programação Orientada a Aspectos (POA). POA surgiu na década de 90 com o objetivo de separar interesses transversais de um software de seus requisitos

básicos [LRM07].

A possibilidade de instrumentar um teste de software sem precisar modificar o código fonte é uma das vantagens obtidas com a utilização de POA. Com POA trabalha-se diretamente ligado a aspectos. Cada aspecto pode ser visto como um interesse transversal ligado ao software. Na prática, para instrumentação de teste, pode-se codificar um aspecto de tal forma que, na execução do mesmo, retornem resultados pertinentes a um determinado teste.

A aplicação dos aspectos pode ser avaliada para instrumentação de teste como: imitadores virtuais de objetos, teste embutido de componentes de software auxiliado por aspectos, verificador de padrões de código, teste baseado em contrato ou instrumentador de código [LRM07]. A utilização de POA facilita a validação da implementação com algumas características da especificação e o teste de restrições.

Com este objetivo destacam-se os trabalhos de Simão *et al.* [RdSSMM05] que apresentam a ferramenta J-FuT como uma ferramenta de automatização de testes funcionais utilizando POA. A ferramenta utiliza a linguagem de aspectos AspectJ para instrumentação do teste. A ferramenta permite que os códigos de teste fiquem separados dos códigos fontes originais. Entretanto a ferramenta não possui automação na codificação do aspecto, que tem que ser codificado pelo testador.

Já Dzidek*et al.* [DBL05] desenvolveram a biblioteca $ocl2j.jar$ que propõe a geração e instrumentação automática de restrições OCL em Java utilizando aspectos para instrumentação.

Richters e Gogolla [RG03] utilizam a ferramenta USE (UML-based Specification Environment) para facilitar a validação e teste de programa considerando as restrições existentes na especificação OCL. Eles utilizam POA para instrumentar os testes, e para monitorar se os resultados da execução da implementação estão de acordo com as restrições definidas na especificação OCL.

Diovaneli [Dio04] também utiliza POA para validação do programa Java com uma especificação, porém o trabalho não é diretamente relacionado a OCL.

1.1 Motivação

Dado o contexto acima destacam-se os seguintes itens que servem como motivação para o presente trabalho:

- A importância da OCL no contexto de software OO. Com o crescente uso da UML para especificação de projetos, a OCL ganha mais importância no teste de software e existem muitos trabalhos que consideram as especificações OCL nos testes e nas atividades de validação.

- Considerar esses tipos de especificações para gerar dados de teste para o programa, auxilia a revelar defeitos relacionados a caminhos ausentes. Teste de programa sem considerar a especificação pode levar a cobertura de todos os elementos requeridos (ou caminhos) sem entretanto identificar um caminho ausente.

- A aplicação de mutação nas especificações é útil considerando que a especificação não está livre de conter defeitos. A utilização da técnica de análise de mutantes pode auxiliar a identificar defeitos nesse contexto.

- A utilização de POA possui algumas vantagens na instrumentação dos testes: com os aspectos pode-se instrumentar o teste de uma classe sem modificar seu código fonte.

- Apesar de existirem trabalhos na literatura que visam a testar restrições OCL e/ou outros critérios funcionais utilizando aspectos, não foi encontrado nenhum trabalho que utiliza POA para também testar restrições OCL que sofreram mutações. Esse aspecto pode levar a revelar defeitos na implementação devido à uma especificação incorreta descrita pelo operador de mutação.

- O custo de teste de mutação diminui consideravelmente com POA, visto que em uma única execução da classe todos os aspectos definidos sobre a especificação e sobre as especificações que sofreram mutação são executados juntamente.

- Unir a abordagem que possibilite revelar defeitos tanto na implementação quanto na especificação durante o teste de programas é bastante útil, e isso ainda não tem sido explorado na literatura.

1.2 Objetivos

Esse trabalho tem como objetivo contribuir para a atividade de teste de programas OO, propondo uma abordagem de teste baseada em mutação de especificação OCL implementada utilizando aspectos. Essa abordagem possui dois usos principais. O primeiro consiste em verificar se a implementação está de acordo com a especificação definida por um contrato para cada classe, para isso a classe correspondente será instrumentada utilizando POA. Executanto o código instrumentado os casos de teste poderão avaliar se a implementação está de acordo com a especificação.

O segundo uso consiste na mutação da especificação pré-definida por um contrato, utilizando um determinado conjunto de operadores de mutação proposto, e o teste da implementação avaliando possíveis defeitos presentes na especificação. A partir das especificações mutantes geradas, aspectos correspondentes são criados e avaliados pelos casos de teste. Após a execução dos casos de teste, uma medida é fornecida, semelhante ao teste tradicional, que permitirá avaliar adequação de um conjunto de dados de teste e, auxiliar a decidir se o programa foi testado o suficiente.

Para permitir a utilização e validação da abordagem proposta, foi implementada uma ferramenta chamada **MuSA** (teste de **Mu**tação baseado em E**S**pecificações OCL e **A**spectos) que auxilia na aplicação da abordagem. A ferramenta tem a finalidade de facilitar o procedimento de teste e assim tornar a utilização da abordagem viável contribuindo para reduzir esforço e custo da atividade de teste.

Em síntese, o objetivo é poder então checar a implementação da classe com relação à especificação. E também revelar defeitos na implementação considerando possíveis problemas na especificação.

1.3 Organização da Dissertação

A presente dissertação segue organizada da seguinte forma: o Capítulo 2 fornece uma revisão teórica da atividade de teste de software. O Capítulo 3 conceitua e diferencia teste de software orientado a objetos, enfatizando a utilização de POA no auxílio a instrumentação. O Capítulo 4 introduz a abordagem proposta nesse trabalho. O Capítulo 5 fornece os aspectos de implementação da ferramenta desenvolvida para validar a abordagem introduzida. O Capítulo 6 descreve o experimento de validação e comparação. O Capítulo 7 descreve as conclusões obtidas, considerações a respeito do trabalho e seus possíveis desdobramentos. O Apêndice A possui esclarecimentos sobre a linguagem orientada a aspectos AspectJ, que é utilizada pela ferramenta MuSA. O Apêndice B contém o esquema XML que representa o formato de entrada das especificações que a ferramenta MuSA utiliza. O Apêndice C apresenta algumas das telas da ferramenta MuSA.

CAPÍTULO 2

TESTE DE SOFTWARE

O desenvolvimento de software com qualidade, é um dos objetivos almejados pela Engenharia de Software (ES). Para obter a qualidade esperada o teste de software é uma das atividades principais.

O teste de software em sentido mais amplo pode ser visto como uma atividade de Verificação e Validação (V&V). Verificação no que se refere à garantia de implementação correta de uma implementação específica. Validação no que diz respeito a conformidade do software com os seus requisitos especificados [Pre06].

> "A atividade de teste de software é um elemento crítico da garantia de qualidade de software e representa a última revisão de especificação, projeto e codificação" [Pre06].

> "Trata-se de uma etapa fundamental do ciclo de desenvolvimento e representa uma importante premissa para alcançar padrões de qualidade no produto criado" [RdSSMM05].

Mesmo com a aplicação de técnicas, métodos e ferramentas propostas pela ES, defeitos na especificação, projeto e codificação do produto podem ser introduzidos. Esses defeitos são responsáveis pelas falhas no comportamento do software. Por isso, é assunto de diversos trabalhos de pesquisa em Engenharia de Software o aprimoramento constante das técnicas, métodos e ferramentas propostas para teste de software.

O processo de teste de software envolve etapas de planejamento e projeto, execução de casos de teste e avaliação dos resultados obtidos.

Na etapa de planejamento e projeto a técnica a ser aplicada deve ser definida. Após essa etapa, os critérios que serão utilizados também devem ser estabelecidos. Os critérios podem ter duas finalidades:

- Seleção de casos de teste: pois estabelecem propriedades que devem ser satisfeitas para elaboração dos casos de teste;

- Adequação de casos de teste: pois podem ser vistos como predicados a serem satisfeitos para avaliar um conjunto de casos de teste e para considerar a atividade de teste encerrada.

Um conjunto de casos de teste satisfaz um critério quando os casos de teste atendem aos requisitos estabelecidos pelo mesmo, ou seja, exercitam certos elementos, denominados elementos requeridos.

Também faz parte da etapa de planejamento/projeto a escolha dos casos de teste que futuramente serão executados. A escolha dos melhores casos de teste é de grande importância, pois os melhores resultados no teste de software dependem muito dos casos de teste executados.

2.1 Terminologia

O teste de software é uma atividade de V & V vista como fundamental para garantia da qualidade. Como padronização da termologia científica esse trabalho tem como base o padrão IEEE 610.12-1990 [iee90] de terminologia que define os seguintes termos:

- *Fault* (defeito): passo, processo ou definição de dados incorreto, como por exemplo, uma instrução ou comando incorreto;

- *Failure* (falha): produção de uma saída incorreta em relação à especificação;

- *Error* (erro): diferença entre o valor obtido e o valor esperado, ou seja, um resultado incorreto é produzido;

- *Mistake* (engano): ação humana que produz um resultado incorreto, com por exemplo, uma ação incorreta tomada pelo programador.

Neste trabalho, defeitos e enganos são associados à causa e, a falha à conseqüência, ou seja, um comportamento incorreto devido a um defeito.

2.2 Objetivos da Atividade de Teste

A atividade de teste tem como objetivo principal revelar defeitos.

> "O objetivo do teste é simplesmente encontrar o maior número possível de erros com uma quantidade de esforço gerenciável aplicada durante um intervalo de tempo realístico" [Pre06].

> "O teste de software pode parecer uma anomalia no processo de desenvolvimento de software. Durante as fases iniciais de definição e desenvolvimento, o engenheiro tenta construir um software a partir de conceitos abstratos em uma implementação concreta. Na fase de teste o engenheiro constrói uma série de casos de teste que tenta "destruir" o software que foi construído" [Pre06].

Myers [Mye04] cita três regras que servem como objetivos para o teste de software:

1. A atividade de teste é o processo de executar um programa com a intenção de descobrir um defeito.

2. Um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um defeito ainda não descoberto.

3. Um teste bem sucedido é aquele que revela um defeito ainda não descoberto.

Também é objetivo dos testes a construção de casos de teste capazes de cobrir diferentes classes de defeitos com o mínimo de tempo e esforço. É válido lembrar que o teste não mostra a ausência de defeitos em um software, apenas pode mostrar que defeitos estão presentes [Pre06].

Apesar de sempre ser o mesmo o objetivo do teste de software encontrar defeitos, os trabalhos nessa área podem ter objetivos mais específicos: definir as melhores técnicas para cada situação, os melhores critérios na aplicação de determinada técnica, a busca pela seleção automática dos melhores casos de teste, etc.

2.3 Fases do teste

É possível identificar fases específicas no processo de teste, cada fase com objetivos distintos. A execução de todas as fases simboliza a validação e verificação do software. Segundo Pressman [Pre06] as fases propostas são assim definidas:

1. Teste de unidade: Corresponde à verificação da menor unidade de software, o módulo; visa a identificar erros de lógica e implementação;

2. Teste de integração: Valida a integração entre os módulos do software; procura por erros de interface entre os mesmos;

3. Teste de validação: Verifica se o produto está de acordo com a especificação dos requisitos do software. É idealizado a partir da execução de uma série de testes de caixa preta. Os resultados dos testes demonstram a conformidade aos requisitos.

4. Teste de sistema: É composto por uma série de diferentes testes com finalidade de exercitar por completo o software. Apesar de ser composto por diferentes testes específicos, todos trabalham para verificar se todos os elementos do software estão adequadamente integrados e executam as funções a eles alocadas. Podem-se citar como exemplos de teste de sistema: teste de recuperação, testes de segurança, teste de estresse, teste de desempenho [Pre06].

2.4 Técnicas de Teste

De acordo com a fase de teste do software e ainda em que contexto o software é projetado e desenvolvido, o testador pode escolher entre algumas técnicas que deseja aplicar. Todas as técnicas têm o mesmo objetivo: encontrar defeitos em um software.

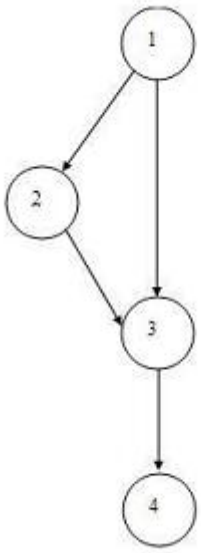
Entretanto a metodologia de cada uma das técnicas, tais como, os critérios apropriados as mesmas são bem diferenciadas.

A técnicas são classificadas geralmente em três grupos: técnica estrutural, técnica funcional e técnica baseada em defeitos.

2.4.1 Técnica estrutural

A técnica presupõe que o testador tem conhecimento da estrutura interna do software incluindo o código fonte.

Os critérios da técnica estrutural, em sua maioria, utilizam uma representação de programa como grafo de fluxo de controle ou grafo de programa. Os grafos possuem os nós que representam a execução de comandos do programa, cada aresta apresenta uma alteração no fluxo de controle, normalmente originada pelos comandos condicionais e de repetição. A Figura 2.1 ilustra um exemplo de um grafo de fluxo de controle de um programa.




```
 void max() {
   float m, max, n;
1    scanf("%f", &m);
1    scanf("%f", &n);
1    max = m;
1    if (n>=m)
2      max = n;
3    printf(%f, max);
4}
```

Figura 2.1: Grafo de Fluxo de Controle (GFC)

Uma das limitações dos critérios estruturais é a existência de caminhos não executáveis. Um caminho é dito não executável se não existe nenhum conjunto de dados de entrada (variáveis de entrada, variáveis globais e parâmetros do programa) que cause sua execução. Por exemplo considere o programa da Figura 2.2. Note que o caminho 1, 2, 4 é não executável e isso não constitui nenhum erro no programa. Um elemento requerido por um critério estrutural é não executável quando todos os caminhos que o exercitam são não executáveis. A existência de caminhos não executáveis impede que o critério estrutural seja sempre 100% satisfeito.

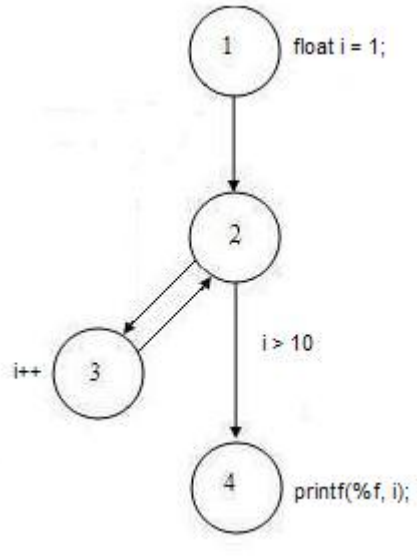


Figura 2.2: Grafo representado caminho não executável

As classes de critérios utilizados na técnica estrutural são: baseados em fluxo de controle [GG75], critérios baseados em complexidade [MB89] e baseados em fluxo de dados [MBV+], [RW85] e [Mal91].

2.4.1.1 Critérios baseados em fluxo de controle

Os critérios baseados em fluxo de controle utilizam somente as caracteríticas de controle de execução do programa para deteminar quais são as estruturas necessárias. São considerados os principais critérios dessa categoria:

- Todos-Nós: exige a execução de todos os nós pelo menos uma vez, ou seja, cada comando do programa seja executado pelo menos uma vez;
- Todos-Arcos (ou Todos-Ramos): considera que cada aresta do programa seja executada ao menos uma vez, ou seja, requer que cada desvio de controle do programa seja executado ao menos uma vez;
- Todos-Caminhos: considera que todos os caminhos do programa sejam executados ao menos uma vez. Apesar de ser o critério mais completo da categoria, o critério Todos-Caminhos é visto como impraticável, pois pode requerer um número infinito de caminhos.

A Tabela 2.1 apresenta os elementos requeridos pelos principais critérios baseados em fluxo e controle do programa *max.c* apresentado na Figura 2.1.

Tabela 2.1: Critérios Baseados em Fluxo de Controle.

Critério	**Exemplo**
Todos-Nós	1, 2, 3, 4
Todas-Arestas	1-2, 1-3, 2-3, 3-4
Todos-Caminhos	1-2-3-4, 1-3-4

Existem outros critérios baseados em fluxo de controle como o critério Boundary-Interior [How87] e a família de critérios K-tuplas requeridas de Ntafos [Nta84].

2.4.1.2 Critérios baseados em fluxo de dados

Os critérios baseados em fluxo de dados baseiam-se no teste das interações que envolvam definições e uso de variáveis no decorrer da execução do programa. Como motivação à utilização dos critérios baseados em fluxo de dados tem-se a indicação de que o teste baseado em fluxo de controle não é eficaz para revelar defeitos até mesmo simples em determinados programas.

Os critérios propostos por Rapps e Weyuker [RW85], como os propostos por Maldonado [Mal91] destacam-se entre os critérios da categoria de critérios baseados em fluxo de dados.

Rapps e Weyuker introduziram uma extensão para o grafo de fluxo de controle chamada de grafo de definição e uso (grafo def-uso).

No grafo def-uso são adicionadas informações com respeito ao fluxo de dados, especificamente: definições de variáveis e utilização dessas variáveis.

Nos critérios baseados em fluxo de dados alguns conceitos se destacam:

- Uma definição de variável (def): ocorre no armazenamento de algum valor em memória, caracterizado por comandos de atribuição, comandos de entrada, ou ainda se a variável é um parâmetro de saída em chamadas de um procedimento. É definido como $def(i)$ o conjunto de variáveis definidas no nó i.

- Um uso computacional ($c - uso$): trata-se da referência ao valor de uma varíavel

afetando diretamente o resultado de uma computação. É definido como $c-uso(i)$ o conjunto de variáveis que possuem um uso computacional no nó i.

- Um uso predicado ($p-uso$): trata-se da referência ao valor de uma varíavel afetando diretamente o fluxo de controle. É definido como $p-uso(i,j)$ o conjunto de variáveis que possuem um uso predicativo no arco (i,j).

- Um $du-caminho$: é um caminho (n_1, ..., n_j,n_k) com relação a uma varíavel (por exemplo x), se x pertence ao $def(n_1)$ e: 1) n_k tem um c-uso de x, (n_1, ..., n_j,n_k) é livre de definição com relação a x e trata-se de um caminho simples, ou seja, todos os nós, exceto possivelmente o primeiro e o último são distintos; ou 2) (n_j, n_k) tem um p-uso de x, $(n_1, ..., n_j)$ é livre de definição com relação x e livre de laços.

A Figura 2.3 ilustra um exemplo de um grafo def-uso.

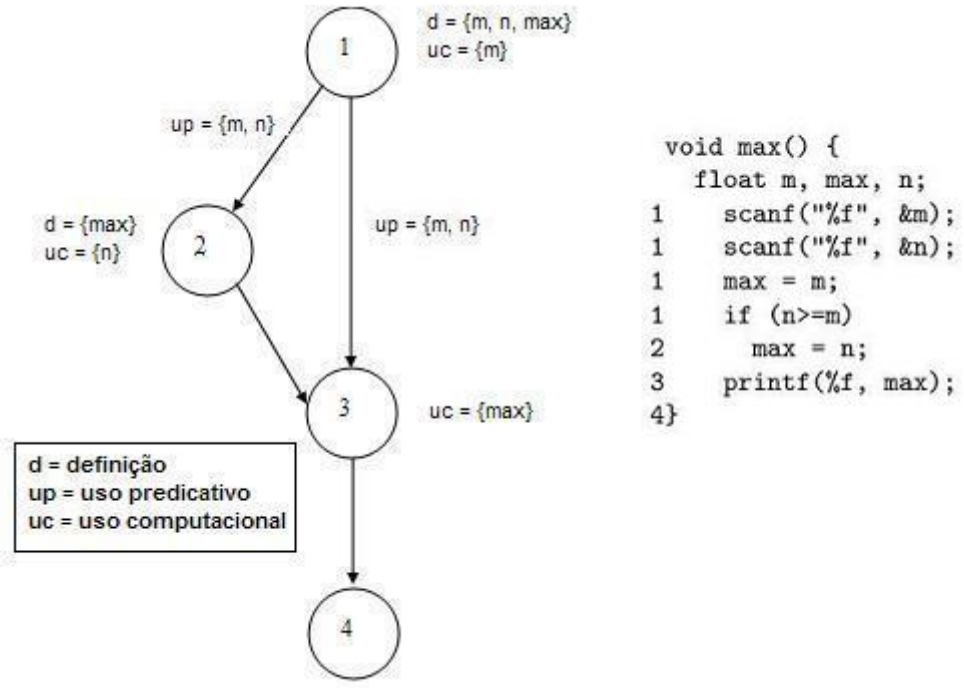


Figura 2.3: Grafo de definição e uso

Com base em todos os conceitos propostos, Rapps e Weyuker definiram uma família de critérios de fluxo de dados [RW85], os principais critérios dessa família são:

- Todas-definições: requer que cada definição de variável seja executada ao menos uma vez por um uso qualquer;

- Todos-usos: requer que todas as associações entre as definições de variáveis e seus usos sejam executados ao menos uma vez, por um caminho no qual a variável não é redefinida;

- Todos-du-caminhos: requer que todos os du-caminhos sejam executados pelo menos uma vez.

Maldonado [Mal91] por sua vez propõe uma extensão ao trabalho de Rapps e Weyuker e introduz a família de critérios Todos-Potenciais-Usos.

Enquanto o trabalho de Rapps e Weyuker considera em seus critérios a ocorrência explicita do uso de uma variável, Maldonado considera que para os critérios serem satisfeitos, os casos de teste devem exercitar não somente os nós com ocorrência explicita do uso de uma variável, mas também os que potencialmente podem ter o uso.

Os critérios potenciais usos são detalhados por Maldonado em [Mal91] e incluem: todos-potenciais-usos, todos-potenciais-usos/du, todos-potenciais-du-caminhos.

2.4.2 Técnica funcional

A técnica funcional, também conhecida como caixa preta, gera dados de teste a partir dos requisitos dos software. A técnica leva em consideração a análise das entradas e saídas do software em teste, sem entretanto considerar informações sobre a sua estrutura interna, o software é visto como uma caixa preta. Alguns dos principais critérios da técnica funcional são sub-itens dessa seção [Pre06] [RdSSMM05].

2.4.2.1 Particionamento em Classes de Equivalência

A partir das condições de entradas de dados identificadas na especificação, divide-se o domínio de entrada de um software em classes de equivalências válidas e inválidas. Com as classes de equivalência definidas, definem-se casos de teste, com a idéia de que um elemento de uma dada classe válida representaria toda a classe válida. Também se faz necessária a seleção de um caso de teste que represente cada uma das classes inválidas.

Alguns autores também consideram o domínio de saída do programa para estabelecer as classes de equivalência.

Com o uso do particionamento é então possível examinar os requisitos sistematicamente, restrigindo o número de casos de teste somente ao necessário.

2.4.2.2 Análise de Valor Limite

Complementa o particionamento de equivalência, tem a mesma idéia de teste de cada uma das classes válidas e inválidas, porém os casos de teste selecionados são mais próximos às, ou estão nas, fronteiras (limites) das classes de equivalência. Entende-se que nesses limites encontra-se um maior número de defeitos.

2.4.2.3 Grafo de Causa-efeito

O grafo de causa-efeito com base na combinação das condições de entrada estabelece os requisitos de teste. Visa a revelar defeitos não apontados pelo particionamento de classes de equivalência ou análise de valor limite. Como o total de combinações das condições de entrada pode ser muito grande, o grafo de causa-efeito pretente auxiliar na escolha de um conjunto de casos de teste, assim como apontar defeitos na especificação.

Nesse critério as saídas possíveis também são levadas em consideração para a combinação de condições.

2.4.3 Técnica Baseada em Defeitos

A técnica de teste de software baseada em defeitos tem como base as informações dos tipos de defeitos mais freqüentes no processo de desenvolvimento de software para derivar os casos de teste que podem ser usados para detectar esses defeitos.

Segundo Binder [Bin99] os tipos de defeitos que um desenvolvedor pode cometer são limitados pelas formas similares utilizadas de se expressar o que se tem na mente, bem como o que se utiliza para desenvolver.

Assim sendo, a variação da linguagem, técnica ou método utilizado para o desenvol-

vimento são base de informação da técnica baseada em defeitos, para definição dos tipos de erros que podem estar presentes no programa e qual o critério ideal a ser considerado.

Semeadura de Erros e Análise de Mutantes (AM) são critérios que se destacam nessa técnica [MBV+]. A seguir o critério AM será detalhado.

2.4.3.1 Critério Análise de Mutantes

O critério Análise de Mutantes (AM) firma-se com base em dois princípios [DLS78]:

- Hipótese do programador competente: assume que os programadores são experientes e tentam escrever software correto ou próximo disso, ou seja, sem erros muito gritantes. Pode-se então afirmar que erros são introduzidos no software através de pequenos desvios sintáticos que não causam erros sintáticos, mas anomalias semânticas que levam o software a um comportamento incorreto [ADH+89];

- Efeito de acoplamento: assume que os erros complexos são originados de erros simples. Assume-se que um caso de teste que é capaz de revelar um erro simples, pode também revelar um erro complexo [DLS78].

O funcionamento do critério AM, partindo de um programa incial P e de um conjunto de casos de teste T, pode ser resumido nos seguintes passos:

1. Geração de um conjunto de mutantes M: através de pequenas alterações ou desvios sintáticos de P gerar mutações do programa. Para maior desempenho, o conjunto de programas mutantes M deve ser adequado para conseguir revelar o maior número de defeitos de P, e ao mesmo tempo, ter pequena quantidade de mutantes, para que seja possível de ser tratado. Os mutantes devem ser gerados a partir de um conjunto de operadores de mutação pré-definidos.

   Os princípios do critério AM citados no início dessa seção são a base para obter-se um conjunto de mutantes adequado: esse conjunto deve possuir defeitos considerados simples (também chamados de *defeitos de primeira ordem*), e conforme o efeito de acoplamento esses defeitos simples podem revelar defeitos complexos.

O número excessivo de mutantes definido nessa primeira etapa, pode resultar em um alto custo ao projeto, podendo inviabilizar a continuidade do mesmo. Para conseguir um número de mutantes adequado, é possível utilizar alguns métodos de redução da quantidade de mutantes. Propostas já foram estudadas sobre métodos de redução de mutantes, sem diminuir o número de defeitos revelados em P: Mutação Aleatória [ABD+79] [MW93], Mutação Restrita [MW93] [WM95] , Mutação Seletiva [ORZ93] e Conjuntos Essenciais de Operadores de Mutação [OLR+96].

2. Execução do programa P com os casos de teste T: consiste em executar o programa P com cada um dos casos de teste verificando o comportamento do programa, se o comportamento for inadequado ou apresentar resultados incorretos para algum caso de teste, então um defeito foi detectado.

3. Execução dos mutantes M com os casos de teste T: consiste em executar cada um dos mutantes M com cada caso de teste t pertecente ao conjunto de casos de teste T. Se o resultado retornado na execução do mutante com o caso de teste for diferente do resultado retornado na execução do programa P, pode se dizer que o M está morto e pode representar um defeito no programa. Caso contrário, se a execução do mutante apresentar resultado igual ao alcançado na execução do programa P, concluí-se que o mutante está vivo, podendo ser equivalente. Um mutante M é equivalente se não existe dado de teste que seja capaz de diferenciar o comportamento de M do comportamento de P, ou seja se P e M computam a mesma função. A presença de mutantes vivos indica que o conjunto de casos de teste T é de baixa qualidade, pois não distingue M de P. O passo de execução dos mutantes pode ser todo automatizado.

   Segundo DeMillo e Offutt [DO91], para que um caso de teste t pertencente ao conjunto de casos de teste T mate um mutante M com relação ao programa P, três condições estão relacionadas:

   - **Alcançabilidade:** O elemento que sofreu mutação no programa m deve ser executado pelo caso de teste t;

- **Necessidade:** Após a execução da mutação o estado do programa m deve diferir do estado do programa P no mesmo ponto de execução;
- **Suficiência:** O resultado da execução de m deve diferir do resultado de execução de P, ou seja, a mutação executada deve refletir até o encerramento da execução do programa.

Concluindo a execução dos mutantes é possível ter um escore de adequação do conjunto de caso de teste T. Esse escore indica se o conjunto de casos de teste é adequado para análise do programa P. O escore (scr) de um conjunto de caso de teste T sobre um programa P é obtido a partir da seguinte fórmula:

$$scr(P,T) = MM(P,T)/(M(P) - EM(P))$$

Sendo que: $MM(P,T)$: representa o número de mutantes mortos pelos casos de teste T; $M(P)$: representa o número de mutantes gerados; $EM(P)$: representa o número de mutantes equivalentes ao programa P. O valor de $scr(P,T)$ é um número entre 0 e 1.

4. Avaliação/Análise dos mutantes: é considerado o passo que mais requer intervenção humana. A partir do escore de mutação obtido é necessário decidir se o teste deve continuar ou não. Caso o escore estiver bem próximo a 1, pode-se decidir por encerrar o programa, avaliando que o conjunto de casos de teste T é suficientemente bom para o programa P. Caso opte por continuar o teste, é necessário avaliar se os mutantes que sobreviveram são equivalentes ou não ao programa original. Novos casos de teste devem ser incluídos ao conjunto de casos de teste para matar os mutantes vivos não-equivalentes. Referente à automatização desse passo, ainda é indecidível determinar a equivalência de um mutante ao programa original via sistema automatizado, sendo necessária a intervenção humana para análise e tomada de decisão.

2.5 Formas de Teste

O teste pode ser aplicado nas diversas fases do desenvolvimento de software. Desde da fase de análise até a fase de implementação [DMJ07].

O teste de software pode ser classificado em duas formas: teste baseado na especificação e teste baseado em programa [How87]. A classificação dos testes considera o nível no qual os critérios de testes são aplicados. Uma breve descrição sobre as duas classificações é apresentada a seguir:

- Teste baseado em especificação: É proposto o teste de software com base nos documentos de especificação. Essa forma de teste é interessante para comprovar a corretude e completude da implementação do que foi especificado. O teste baseado em especificação, pode ser prejudicado em situações nas quais a especificação não é definida de forma cuidadosa e coerente. Os testes baseados em especificação geralmente consideram diferentes modelos tais como modelo de contratos, de objetos, de comportamento, etc.

  O contrato é um modelo de especificação que vem sendo atualmente bastante explorado no teste de software e é de particular importância para o presente trabalho. Um contrato é uma especificação de software que define obrigações entre as partes (fornecedor do serviço e cliente do serviço), assim como os resultados do seu relacionamento. O contrato é geralmente definido por *assertivas*. As assertivas são utilizadas para a geração de casos de teste. Três tipos de assertivas são consideradas na técnica de projeto por contrato:

  - *Pré-condições:* condições testadas antes da chamada da função fornecida, representa as obrigacões que o cliente deve cumprir para utilizar o serviço fornecido;
  - *Pós-condições:* condições testadas após a execução do serviço fornecido;
  - *Invariantes:* condições que não devem ser alteradas com a execução do serviço externo.

O teste baseado em contrato tem sido aplicado para o teste de *Web Services* [MZ05] e de componentes [JHS+05].

- Teste baseado em programa: O teste baseado em programa vem reduzir algumas limitações do teste baseado em especificação, por exemplo: a não quantificação da atividade de teste, a não garantia que partes críticas do programa sejam executadas e, ainda o fato de que a especificação do software está sujeito à inconsistência. Na seção anterior os diferentes critérios de teste foram ilustrados e utilizados para o teste baseado em programa.

2.6 Teste de Regressão

A evolução do software é constante seja por correções de defeitos, ou por causa da evolução proposta pelas novas implementações. É fato que algumas alterações propostas para determinadas unidades do software podem resultar em comportamentos anormais de outras unidades dependentes da mesma. Uma boa implementação do desenvolvedor do software aliada a uma especificação adequada, permite que se tenha controle sobre cada unidade que sofre alteração de comportamento.

Entretanto, avaliando que nem sempre tem-se a especificação adequada, e que a implementação do software também não corresponde idealmente ao que foi especificado, a Engenharia de Software propõe a utilização de Testes de Regressão (TR) como base de validação periódica do software que sofre freqüentes alterações.

O TR é um teste normalmente automatizado, exercitado regularmente em busca de encontrar erros originados de manutenções em um software. A cada nova manutenção no software, o TR pode ser realizado com diversos casos de teste.

Os testes de regressão são importantes quando o software passa para a fase de manutenção contínua, no qual sofre alterações constantes e que normalmente influenciam mais de um módulo [dPPF03].

> "Uma bateria de teste de regressão consiste em um conjunto selecionado de testes de boa cobertura, que é executado periodicamente, de forma auto-

matizada. Caso essa bateria seja muito grande, pode-se utilizar uma bateria menor com testes selecionados, para uso mais freqüente, testando-se a bateria completa em intervalos maiores" [dPPF03].

"O teste de regressão é uma estratégia importante para reduzir efeitos colaterais. Deve-se executar testes de regressão toda vez que uma mudança importante é feita no software (inclusive a integração de novos componentes)" [Pre06].

Com os testes de regressão é possível também verificar se a documentação do produto corresponde ao seu comportamento atual, assim como se os procedimentos e os casos de teste continuam válidos.

2.7 Considerações Finais

De forma resumida esse capítulo apresenta toda uma organização de definições relacionadas ao teste de software. A estruturação de diferentes técnicas para teste mostra que muito já foi estudado e o resultado desse estudo é base para os novos trabalhos que surgem. A AM, critério baseado em defeitos, tem sido encontrada em estudos experimentais [WMM94] como um dos mais eficazes em termos do número de defeitos revelados e por isso é o foco desse trabalho, tendo sido apresentado com mais detalhe.

O próximo capítulo trata do teste voltado a software desenvolvido no paradigma de orientação a objetos (OO), introduzindo novos conceitos relacionados ao teste neste contexto.

CAPÍTULO 3

TESTE DE SOFTWARE ORIENTADO A OBJETOS

A variação de contexto no desenvolvimento de software, em muitos casos, obriga a aplicação diferenciada de determinada técnica ou metodologia.

Por exemplo, a utilização do paradigma de orientação a objetos (OO) que se diferencia rigorosamente do desenvolvimento de software procedimental, fez com que as técnicas tradicionais de teste de software sofressem adaptações que correspondessem à necessidade desse paradigma específico.

Segundo Pressman [Pre06], apesar de se ter o mesmo objetivo fundamental quanto ao teste, o teste em OO utiliza diferentes estratégias e táticas.

Pressman [Pre06] descreve que utilizando o paradigma OO o domínio do problema é caracterizado como um conjunto de objetos que possuem propriedades e comportamentos específicos e se comunicam por meio de mensagens.

Utilizando a orientação a objetos, há maior facilidade no desenvolvimento, visto que promove o reuso e torna a manutenção mais facilitada graças ao ocultamento da informação. Porém, Vincenzi *et al.* [VDDM07] alertam que o uso da orientação a objetos não é capaz de garantir a correção de um programa por si própria, visto que ela não impede que os desenvolvedores cometem enganos durante o desenvolvimento de um produto de software. O que se tem observado é que as características principais da orientação a objetos introduzem novas fontes de defeitos de modo que atividades de Verificação, Validação e Teste (VV&T) são de fundamental importância no contexto de programas OO.

O fato é que com o paradigma OO surgiram novos conceitos (encapsulamento, herança, polimorfismo, etc). Conceitos esses que se tornaram novos desafios para os testes de software.

"De forma geral, os conceitos e técnicas aplicados ao teste de programas procedimentais podem ser aproveitados para o teste de programas OO, po-

rém a ênfase e a efetividade das várias técnicas diferem nessa nova abordagem" [RdSSMM05].

3.1 Fases do Teste de Software OO

Existem abordagens diferentes sobre as fases de teste de software OO. Duas são descritas a seguir.

Na concepção apoiada por Harrold e Rothermel [HR94], o método é a menor unidade a ser testada. O teste de unidade corresponde ao teste intra-método. O teste de integração é representado por testes inter-métodos e intra-classe.

Já para Pressman [Pre06], apesar de considerar que as operações (métodos) de uma classe são a menor unidade testável, devido a essas operações poderem fazer parte de um certo número de classes diferentes, a tática aplicada ao teste de unidade precisa modificar-se, não podendo ser a operação a base do teste unitário.

Pressman define:

> "O teste de classe para o software OO é equivalente ao teste de unidade para software convencional. Diferentemente do teste de unidade do software convencional, que tende a focalizar o detalhe algorítmico de um módulo e os dados que fluem através da interface do módulo, o teste de classe para software OO é guiado pelas operações encapsuladas e pelo estado de comportamento da classe" [Pre06].

Em ambas abordagens, o teste de sistema tem o mesmo significado e a mesma forma de aplicação.

Neste trabalho uma abordagem similar a de Harrold e Rothermel [HR94] é adotada. Os tipos de testes (de unidade e de integração) utilizados são apresentados na Tabela 3.1.

Tabela 3.1: Fases de Testes OO X Tipos de Testes Adotados nesse trabalho

Fases de Testes	**Tipos de Testes Adotados**
Unidade	intra-método
Integração	inter-métodos, intra-classe e inter-classe
Sistema	toda aplicação

Os testes da Tabela 3.1 podem ser assim descritos:

- Intra-Método: corresponde ao teste de unidade do paradigma OO, propõe o teste individual de cada método;
- Inter-Método: corresponde a parte do teste de integração do paradigma OO, testa a integração entre a chamada de um método público com os demais métodos internos da classe;
- Intra-Classe: corresponde a parte do teste de integração do paradigma OO, testa a integração entre métodos públicos da mesma classe. Como os métodos podem ser chamados em ordem aleatória, é importante a chamada dos métodos públicos em ordem indeterminada a fim de avaliar o comportamento dos objetos;
- Inter-Classe: também corresponde ao teste de integração do paradima OO, testa a integração entre métodos públicos de classes diferentes, a fim de avaliar o comportamento na integração de objetos de classes diferentes.

3.2 Tipos de Defeitos em Software OO

É fato que a programação orientada a objetos pode reduzir a ocorrência de alguns tipos de defeitos encontrados na programação procedimental (por exemplo: a diminuição de linhas por método, propõe a diminuição da ocorrência de defeitos de fluxo de controle). Entretanto, a programação orientada a objetos, a partir da aplicação de seus conceitos, introduz outros tipos.

Nos sub-itens dessa seção, são apresentados alguns dos defeitos mais prováveis que podem ocorrer, com a aplicação de determinados conceitos da programação OO. Esse conteúdo foi extraído de Vincenzi *et al.* [VDDM07].

3.2.1 Com Encapsulamento

O encapsulamento se refere ao tratamento da visibilidade dos atributos e métodos da classe. A partir do encapsulamento, é possível controlar o acesso de classes externas a

atributos e métodos que são de acesso indesejável.

É a partir do encapsulamento que se eliminam dependências indesejáveis de uma classe servidora (de atributos e métodos) com uma classe cliente.

Para o teste de software, o encapsulamento vem como um problema no que diz respeito ao fato de não se ter conhecimento em determinados momentos do estado de determinados atributos de um objeto, atributos esses com visibildiade restrita.

Como solução ao problema identificado com esse conceito pode ser aplicada a implementação de todos os métodos *get* e *set* a todos os atributos da classe [Har00], ou ainda, a utilização de recursos de reflexão computacional [RM98].

3.2.2 Com Herança

O mecanismo de herança é indispensável na orientação a objetos, trata-se da reusabilidade via o compartilhamento de características presentes em classes ancestrais. É possível compartilhar atributos e métodos de uma superclasse a uma subclasse.

No entanto, Binder [Bin99] afirma que a aplicação da herança enfraquece o encapsulamento e pode ser responsável pela criação de um risco similar ao uso de variáveis globais em programas procedurais. Ao se implementar uma classe com o uso de herança é necessário ter a compreenssão dos detalhes de implementação das classes ancestrais. Grandes hierarquias de herança podem também ser responsáveis por provocar enganos freqüentes de desenvolvedores, gerando defeitos causadores de falhas.

O assunto de testes sobre o mecanismo de herança, já foi tema de alguns trabalhos [PCM96], [HMF92] e [MS01]. Algumas conclusões desses trabalhos foram:

- a utilização da herança pode apresentar uma falsa impressão que as subclasses de superclasses testadas não precisam ser testadas;
- um método mesmo que integralmente herdado de uma superclasse, deve ser retestado na subclasse.

E ainda, os trabalhos resultaram no surgimento de uma estratégia incremental hierárquica, que idealiza identificar quais métodos herdados necessitam de novos casos de

teste para serem testados, e quais podem ser retestados com os mesmos casos usados nas superclasses. Segundo os autores Harrold *et al.* [HMF92] e McGregor e Sykes [MS01] o uso de tal estratégia reduz significamente o esforço de teste.

3.2.3 Com Herança Múltipla

Está associada à capacidade de herdar características de duas ou mais superclasses que podem conter, ou não, atributos e métodos com o mesmo nome.

Entre possíveis defeitos originados da herança múltipla, um deles pode ser identificado no caso de duas superclasses com um método com mesmo nome, poderem executar na subclasse hora a chamada do método de uma superclasse e hora a chamada do método de outra superclasse (conforme modificações sintáticas). Isso provavelmente resulta em problemas no que diz respeito aos casos de teste que possivelmente não serão os mesmos necessários para validação dos dois métodos herdados.

3.2.4 Com Classes Abstratas e Genéricas

Classes abstratas, em geral, representam somente uma interface sem nenhum método implementado. Não é possível instanciar um objeto a partir de uma classe abstrada. A classe abstrata tem a finalidade de reuso, ou seja, a utilização da mesma como superclasse de uma nova classe a ser implementada.

Para os testes de software o resultado negativo é que não se tem como testar a classe abstrata pela instanciação dos objetos originados da mesma. O teste da classe abstrata só é possível sobre o objeto instanciado de uma de suas subclasses.

Já as classes genéricas não são abstratas, porém tem a mesma qualidade de facilitar o reuso. A partir de uma classe genérica é possível a derivação de diversas subclasses. Quanto à instanciação de um objeto, é possível definir um objeto como sendo de uma classe genérica, porém na sua instanciação definir que ele é uma instância de uma subclasse da classe genérica.

Em termos de teste de software o fato de o objeto de classe genérica ser instanciado em tempo de execução a partir de uma subclasse da classe genérica é prejudicial, visto

que para definição dos casos de teste, fica indecidível definir quais os melhores casos de teste a serem aplicados.

3.2.5 Com Polimorfismo

O polimorfimo caracteriza a possibilidade de um mesmo nome ser utilizado para representar instâncias de várias classes. A aplicação do polimorfismo é muito comum em conjuntos hierárquicos formados. Apesar de ser um mecanismo muito útil e fundamental na OO, propicia riscos de possíveis defeitos no software, entre eles: indecibilidade no acoplamento dinâmico, extensibilidade de hierarquias, containers heterôgeneas e *type casting* [DMJ07].

3.3 Teste baseado em programas OO

No contexto de teste baseado em programas OO, destacam-se os estudos realizados na adequação e inovação dos critérios estruturais baseados em fluxo de controle e em fluxo de dados e, dos critérios baseados em mutação, antes aplicados no paradigma procedimental.

Harrold e Rothermel [HR94] estenderam o teste de fluxo de dados para o teste de classes. Eles definiram pares (associações) de definições e uso que permitem avaliar relações de fluxo de dados em programas OO. Os pares definidos são de acordo com o nível de teste adotado:

- pares Def-Uso Intra-Método: que permite testar pares definição-uso dentro de um único método;
- pares Def-Uso Inter-Método: que ocorrem quando métodos dentro do contexto de uma invocação interagem e a definição de uma variável dentro dos limites de um método alcança um uso dentro dos limites de outro método chamado direta ou indiretamente por um método público;
- pares Def-Uso Intra-Classe: que ocorrem quando seqüencias de métodos públicos são invocados.

Considerando principalmente o teste intra-método, Vincenzi *et al.* [VMWD05] a partir do estudo de diversos trabalhos sobre critérios baseados em fluxo de controle e fluxo de dados, e da linguagem de código objeto Java *(bytecode)*, propuseram o desenvolvimento de uma ferramenta para avaliar códigos compilados em Java, denominada JaBUTi, detalhada a seguir.

3.3.1 Ferramenta JaBUTi

A JaBUTi *(Java Bytecode Understanding and Testing)* é uma ferramenta utilizada para instrumentação da técnica estrutural. É aplicada sobre o software desenvolvido com a linguagem Java. Uma das suas características diferenciais, é que sua análise estática é executada sobre o código objeto em Java, ou seja, sobre os *bytecodes*.

A ferramenta de teste JaBUTi efetua teste baseado em fluxo de dados e fluxo de controle de programas OO. Ela implementa os critérios de teste intra-métodos, sendo quatro de fluxo de controle (Todos-Nós-ei, Todos-Nós-ei, Todas-Arestas-ei, Todas-Arestas-ei) e quatro de fluxo de dados (Todos-Usos-ei, Todos-Usos-ed, Todos-Pot-Uso-ei, Todos-Pot-Uso-ed).

A descrição de exigência de cada critério baseado em fluxo de controle da JaBUTi pode ser definida assim:

- Todos-Nós *ei*: Exige a cobertura de todos os comandos não relacionados com o tratamento de exceção;
- Todos-Nós *ed*: Exige a cobertura de todos os comandos relacionados com o tratamento de exceção;
- Todas-Arestas *ei*: Exige a cobertura de todos os desvios condicionais do método, desvios esses, não decorrentes do lançamento de uma exceção;
- Todos-Nós *ed*: Exige a cobertura de todos os desvios de execução decorrentes do lançamento de uma exceção.

A descrição de exigência de cada critério baseado em fluxo de dados da JaBUTi pode ser definida assim:

- Todos-Usos *ei*: Exige a cobertura de todas associações definição-uso não relacionados com o tratamento de exceção
- Todos-Usos *ed*: Exige a cobertura de todas associações definição-uso relacionados com o tratamento de exceção
- Todos-Pot-Usos *ei*: Exige a cobertura de todas potenciais-associações definição-uso não relacionados com o tratamento de exceção.
- Todos-Pot-Usos *ed*: Exige a cobertura de todas potenciais-associações definição-uso relacionados com o tratamento de exceção.

A JaBUTi tem uma fácil visualização dos blocos de códigos que são marcados com um peso, como uma das suas características principais. Os pesos servem para indicar qual o grau de cobertura de cada bloco de código. Isso facilita a geração de casos de teste, de modo a maximizar a cobertura em relação aos critérios de teste. Estes blocos mostram de forma inteligente qual o requisito de teste que, se coberto, aumentaria de forma considerável a cobertura em relação ao critério considerado.

A JaBUTi ainda permite que os requisitos de teste de cada um de seus critérios possam ser visualizados no *bytecode*, código fonte e no grafo de cada método de cada uma das classes em teste. Diferentes cores são associadas para indicar seus pesos.

Pela JaBUTi o testador ainda pode avaliar a cobertura de todo projeto em relação a cada um dos critérios de teste. Essa informação pode ajudar o testador a decidir se cada critério já atingiu o seu objetivo, e ainda possibilitar conforme a avaliação do testador a utilização de um critério mais forte, capaz de ajudar na evolução do conjunto de teste.

Métricas estáticas que avaliam a complexidade das classes dos softwares testados, junto com heurísticas de particionamento de software que facilitam a localização dos defeitos, fazem parte das opções da JaBUTi.

Segundo Vincenzi [Vin04], entre as atividades executadas pela JaBUTi para realizar a análise de cobertura, pode se citar: instrumentação de arquivos *.class*, coleta de informações durante a execução do software, e apresentação do quanto cada método de todas as classes foram testados em conformidade com os critérios de teste disponíveis.

3.4 Teste de especificação OO

Os trabalhos que realizam o teste de especificação em OO utilizam diferentes diagramas tais como o de caso de uso [Heu01], [BG02] e [CCJ03], modelos de comportamento [VJ03] e testes baseados em contrato [VJ03] e [JHS+05] por exemplo.

Visando o desenvolvimento de critérios de testes que utilizem os diferentes tipos de diagramas de modelos da OO para auxiliar na geração de caso de teste, Chaim *et al.* [CCJ03] definem uma série de critérios de teste para realizar teste de cobertura em especificações UML, com enfâse em diagramas de caso de uso. A idéia base dos critérios de Chaim *et al.* é assegurar que as interações entre casos de usos e entre atores e casos de usos sejam executadas, contribuindo com mais precisão ao problema que está sendo modelado.

Em teste de especificação baseados em contratos, trabalhos se destacam com a utilização da linguagem de especificação OCL *(Object Constraint Language Specification)* [OMG08]. OCL é uma linguagem de expressão que pode ser usada para especificar invariantes, pré-condições, pós-condições e outros tipos de restrições que a linguagem UML não representa a contento. Apesar de possuir algumas limitações, a OCL é hoje a linguagem de expressão mais usada para especificar formalmente modelos orientados a objetos [AS05].

Aertryck e Jensen [VJ03] apresentam uma ferramenta chamada UML-Casting para gerar casos de teste baseados em diagramas de estado da UML. Desses diagramas, restrições OCL são obtidas e cobertas pelos dados de teste gerados.

Jiang *et al.* [JHS+05] apresentam uma proposta de mutação de contratos para teste voltado a componentes de software. A mutação é feita sobre os contratos definidos para os componentes. A idéia se justifica, entre outras razões, pois se tratando de componentes na maioria das vezes o código fonte não está disponível.

Quanto à definição de contratos, Jiang *et al.*, buscando uma linguagem de especifi-

cação mais simples, ao invés de utilizarem a linguagem OCL ou a JML (Linguagem de Modelagem Java), utilizam uma linguagem própria estendida da linguagem de definição de *Enterprise Java Beans (EJB)*. A linguagem, a princípio, assemelha-se às demais linguagens de especificação de contrato, trabalhando basicamente com pré-condições, pós-condições e invariantes.

Jiang *et al.* [JHS+05] utilizam seu próprio conjunto de operadores de mutação denominados CBMO (operadores de mutação baseado em contratos) conforme a Tabela 3.2:

Tabela 3.2: Operadores de Mutação Utilizados.

Nome	**Especificação**	**Regras**	
		Original	Mutante
CN	Negação do contrato	P == > <	!P != <= >=
CE	Mudança de condição	pré-condição pós-condição*	pós-condição pré-condição
PW	Enfraquecimendo da pré-condição	== > < P>=Q P<=Q forall *and*	>=, <= >=, != <=, != P>=Q-constante P<=Q+constante exists *or*
PS	Fortalecimento da pós-condição	>= <= != P>Q P<Q exists *or*	>, == <, == >, < P>Q+constante P<Q-constante forall *and*
CS	Contrato Furado	pré-condição pós-condição	TRUE FALSE
* = (Sem considerar "Old" e "Result")			

3.5 Teste de programas OO utilizando a especificação

O teste de programa OO pode ser realizado com o auxílio de alguma especificação, por exemplo um contrato.

Benattou *et al.* [BBH02] geram dados de teste de especificações OCL baseadas no particionamento em classes de equivalência. No trabalho, Benattou *et al.* analisam individu-

almente cada método e define subconjuntos de estados que o programa deve apresentar. Cada subconjunto é representado por restrições que podem ser extraídas da especificação OCL e refletem no comportamento do programa naquele estado.

Aicherming e Salas [AS05] exploram a geração de dados de teste como um problema de satisfação de restrições. As restrições, na forma normal disjuntiva (DNF), são obtidas de uma especificação OCL e também de especificações que sofreram mutação. Contudo, eles não introduzem um conjunto de operadores de mutação, nem uma ferramenta. O trabalho é voltado ao desenvolvimento de um algoritmo capaz de gerar dados de teste suficientes para cobrir as restrições geradas a partir da mutação das especificações.

Gogolla *et al.* [GBR03] descrevem uma ferramenta chamada USE (UML-based Specification Enviroment) para checar inconsistências em diagramas de classe com base na OCL.

Já a ferramenta SPACES (Specification bAsed Component tESter) [BLM+07] gera dados de teste para componentes em Java a partir de diagramas em UML e restrições OCL.

3.6 Teste de programas OO apoiados por POA

A Programação Orientada a Aspectos (POA) surgiu na década de 90 com o objetivo de separar interesses transversais de um software, de seus requisitos básicos [LRM07]. A POA possibilita a implementação de módulos isolados, denominados aspectos, que são capazes de afetar de forma transversal vários módulos de um software.

A POA em seu surgimento trouxe um desafio ao teste de software, que é o da adequação de grafos, critérios e estratégias de teste para programas desenvolvidos em POA. Porém ao mesmo tempo, graças a sua possibilidade de permitir com que o código fonte OO de um software consiga ser verificado de forma automatizada, sem alterações no próprio código fonte, tornou-se alvo de interesse de diversos trabalhos de pesquisa sobre teste de software OO. Ou seja, com POA é possível instrumentar o código fonte OO em um arquivo separado sem modificar nada do mesmo.

A POA ainda possibilita que em uma única execução de uma classe diversos aspectos

sejam executados conjuntamente. No caso do teste de software, isso pode ser utilizado para diminuir o custo do software, em situações que diversas instrumentações devem ser executadas sobre a mesma classe.

Lemos *et al.* [LRM07] apontam as seguintes formas de aplicação de aspectos como apoio a teste estrutural de software OO:

1. Como imitadores virtuais de objetos: uma classe desenvolvida entre um conjunto de classes representada em um diagrama, tem seu próprio conjunto de teste a ser executado. Conjunto esse que pode exigir a chamada de um método de uma outra classe. O fato das demais classes do software não terem sido desenvolvidas não elimina a necessidade da classe ser testada. Com OO já é possível simular um método ainda não desenvolvido com a implementação de um método só para teste na classe que faz a chamada. A vantagem de se fazer isso em aspecto é que sem modificarmos o código fonte em teste, pode-se implementar um aspecto que simule o comportamento desejado pelo método ainda não implementado.

2. Como teste embutido e componentes de software auxiliado por aspectos: para componentes que possuem responsabilidades bem definidas, é possível com aspectos implementar casos de teste que testem o funcionamento dessas responsabilidades. A partir de um contrato que acorde todos os valores que podem ser definidos como entrada e saída do componente, tanto o desenvolvedor como o integrador podem viabilizar via aspectos testes essencias para comprovar o funcionamento do componente.

3. Como verificador de padrões de código: POA pode ser utilizada para testes OO de boas práticas de programação e uso adequado de padrões de projeto. Por exemplo, é possível desenvolver um aspecto que verifique se todos os atributos que estão recebendo um valor, possuem um método *set* definido, conforme regra de boa prática de programação OO. Ou ainda, pode se verificar se um determinado método está sendo invocado de determinada classe, e caso não, apontar erro de teste.

4. Como teste baseado em contrato: conforme já definido na Seção 2.5 é possível

efetuar teste de contrato. Sendo que utilizando POA, é possível implementar um aspecto que implemente as assertivas necessárias para o controle de pré-condições, pós-condições e invariantes, sem precisar alterar nada no código fonte.

5. Como instrumentador de código: é possível com POA a implementação de aspectos que controlem informações gerenciais de instrumentação do código fonte na execução do mesmo. Ou seja, pode-se guardar informações como: tempo de execução, valor de determinados atributos, sequência da chamada de métodos, etc.

Exemplificando a utilização de POA no teste de programas Java, Simão *et al.* [RdSSMM05] apresentam a ferramenta J-FuT para automatizar o teste funcional: particionamento de classe de equivalência, análise de valor-limite e teste funcional sistemático.

J-FuT utiliza a tecnologia POA, através da linguagem de aspectos AspectJ para instrumentação do teste. Classificada como "teste baseado em contrato", conforme classificação apontada por Lemos *et al.* [LRM07] quanto à utilização de POA, a ferramenta permite que os códigos de teste fiquem separados dos códigos fontes originais.

A ferramenta J-FuT possui configuração dos critérios de cobertura via arquivos XML que podem ser configurados conforme necessidade de cobertura do testador, definindo entre outros: o número máximo de casos de teste a ser executado e mínimo de cobertura de teste.

É necessário que o testador informe a J-FuT qual o código a ser testado, e quais dados de teste devem ser avaliados (em códigos gerados pela ferramenta JUnit). Também fica a cargo do testador codificar cada aspecto a ser criado a um método em teste e cada um dos requisitos do teste.

Outros exemplos da utilização de POA como instrumentação de teste são encontrados nos trabalhos [DBL05], [RG03] e [Dio04].

Dzidek*et al.* [DBL05] desenvolveram a biblioteca *ocl2j.jar* que utiliza a especificação OCL como base para o teste de programas em Java utilizando POA. A *ocl2j.jar* busca as restrições definidas nos contratos em OCL, e as instrumenta como assertivas em aspectos utilizando a linguagem AspectJ.

Richters e Gogolla [RG03] estenderam a ferramenta USE (UML-based Specification Environment) para facilitar a validação e teste do programa considerando as restrições da especificação OCL. Eles utilizam POA para instrumentar os testes, e para monitorar se os resultados da execução da implementação estão de acordo com as restrições OCL.

Diovaneli [Dio04] também utiliza POA para validação de programa em Java com uma especificação, porém o trabalho não é diretamente relacionado a OCL.

3.7 Considerações Finais

A aplicação do paradigma OO traz diversas vantagens no desenvolvimento de software. Ao mesmo tempo apresenta desafios diferenciados para o processo de validação e verificação do sofware.

O presente capítulo apresentou conceitos e trabalhos realizados direcionados a teste de programas OO. A importância da especificação no teste OO é evidente, e os trabalhos realizados utilizando OCL [AS05], [GBR03], [BLM+07], [DBL05] e [RG03] justificam a importância dada a essa linguagem de especificação para o teste de software OO.

A mutação da especificação também é assunto de alguns trabalhos apresentados [JHS+05] e [AS05], a justificativa para isso é a necessidade de teste da própria especificação [JHS+05] e também do programa [AS05].

POA foi descrita como uma tecnologia que pode auxiliar de diferentes formas o teste estrutural de software OO [LRM07]. Os trabalhos apresentados que utilizam POA para a instrumentação dos testes [DBL05], [RG03], [Dio04] e [RdSSMM05] justificam sua utilização pela possibilidade de instrumentar uma classe sem modificar seu código fonte, e se beneficiam da possibilidade de executar vários aspectos em uma única execução de uma classe.

Considerar a especificação para testar uma particular implementação é muito importante. Devido ao uso crescente dos modelos UML, a OCL ganhou importância e muitos trabalhos consideram especificação OCL nas atividades de teste e validação. Com relação a esses trabalhos duas direções de pesquisa podem ser observadas. A primeira é checar a satisfação de restrições OCL durante a execução (ou geração) do código. Nessa direção os

trabalhos que usam POA apresentam vantagens, pois a instrumentação é mais eficiente e não intrusiva.

A segunda direção é a geração de dados de teste a partir das restrições OCL. Geralmente essa geração é guiada por um critério de teste, tal como o critério de particionamento de classes de equivalência e teste de mutação. A utilização de um critério para guiar essa atividade pode oferecer uma medida de cobertura e avaliação dos dados gerados. Entretanto esses trabalhos geralmente não implementam uma ferramenta.

É interessante considerar ambas direções de trabalho para que isso auxilie a revelar defeitos junto ao programa implementado e/ou na especificação. Com esse objetivo é introduzida uma abordagem descrita no próximo capítulo.

CAPÍTULO 4

ABORDAGEM DE TESTE BASEADA EM ASPECTOS E MUTAÇÃO DE ESPECIFICAÇÕES OCL

Neste capítulo é introduzida uma abordagem para o teste de software OO. Ela propõe a utilização da especificação OCL e da POA para o teste da implementação e da própria especificação. Dois usos para a abordagem são propostos:

1. Uso 1: a geração de dados de teste a partir da especificação para o teste da implementação; e

2. Uso 2: validação da especificação e da implementação por meio da análise de especificações mutantes.

De forma objetiva pode-se dizer que o trabalho busca por defeitos contidos em uma implementação com relação ao que foi especificado, e também busca defeitos na própria especificação OCL que podem estar refletidos na implementação. A abordagem é ilustrada na Figura 4.1.

A abordagem considera as pré e pós-condições das especificações OCL para o teste de uma determinada classe. Para cada especificação um aspecto é gerado correspondendo à instrumentação do teste. Sobre cada dado de teste informado, a classe e os aspectos são executados e informações sobre o teste são armazenadas. Os resultados obtidos são base de avaliação se a implementação da classe está de acordo com a especificação.

Para buscar por defeitos na especificação, a especificação OCL passa por um processo de mutação, no qual especificações modificadas são implementadas e instrumentadas para o teste através de aspectos. Esses aspectos são executados com a classe sobre os dados de teste, e os resultados dessa execução podem apontar por defeitos na especificação e consequentemente no código.

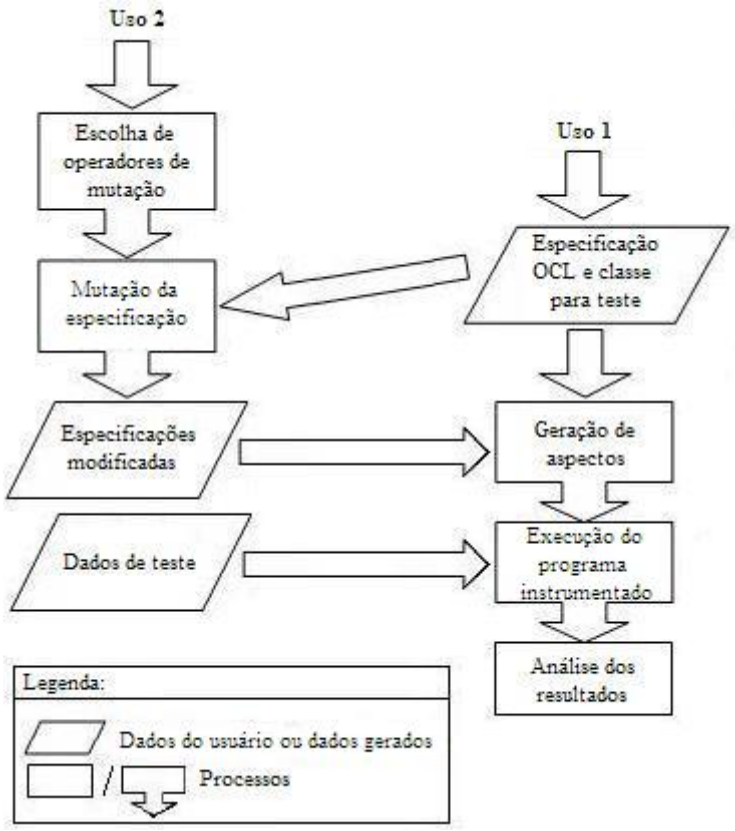


Figura 4.1: Representação gráfica da abordagem nos dois usos

As próximas seções desse capítulo descrevem as tecnologias utilizadas na abordagem e ainda apresentam passo a passo cada um dos usos sugeridos.

4.1 Especificações OCL

A OCL é uma linguagem de expressão muito utilizada para detalhar diagramas propostos pela UML. Um subconjunto da OCL é considerado nesse trabalho, subconjunto esse que descreve para cada método da classe as pré e pós-condições existentes. O subconjunto foi definido com o objetivo de reduzir a complexidade da especificação a ser interpretada pela abordagem. O Código 4.1 ilustra a gramática base para o subconjunto da OCL utilizado na proposta.

Pode se observar que esse subconjunto não inclui operações com *collections* e nem comandos de laço. Não trabaha com chamada a funções e nem tipos enumerados. Trabalha basicamente com os tipos string, números inteiros, de ponto flutuante (reais) e booelanos.

O Código 4.2 representa um arquivo OCL válido conforme a gramática fornecida. O

```
method_list := ("context" name "::" name "(" param_list ")" ":" type)*
       specification
param_list := (name type)*
specification :=  preCond
                  posCond
preCond := "pre:" exp
posCond := "post:" exp
exp :=  lExp
ifExp := "if" exp "then"
              exp
         "else"
              exp
         "endif"
lExp := rExp (lOper rExp)*
rExp := sExp (rOper sExp)?
sExp := elem (aOper elem)?
elem := uExp (aOper uExp )*
uExp := uOper elem  | elem
elem := literal | "(" exp ")" | ifExp
lOper := "and" | "or"
rOper := "=" | ">" | "<" | ">=" | "<=" | "<>"
aOper := "+" | "-" | "*" | "/"
uOper := "not"
literal := string | number | boolean | real | integer | name
boolean := "TRUE" | "FALSE"


()* significa 0 ou mais
()? significa que é opcional
```

Código 4.1: Subconjunto utilizado da OCL

código trata das condições especificadas para o método *type*() da classe *TriTyp*. Esse método não apresenta valor de retorno e tem como finalidade definir ao atributo inteiro *trityp* que formato corresponde ao triângulo formado pelos lados definidos nos atributos inteiros i, j e k.

```
context TriTyp::type():void
  pre:
  post: if ( (self.i<=0) or (self.j<=0) or
        (self.k<=0) or (self.i>=(self.j+self.k))
        or (self.j>=(self.i+self.k)) or
        (self.k>=(self.i+self.j)) then
        self.trityp=4 else if ((self.i=self.j)
        and (self.j=self.k)) then self.trityp=3
        else if ((self.i=self.j) or
        (self.i=self.k) or
        (self.j=self.k)) then self.trityp=2 else
        self.trityp=1 endif endif endif
```

Código 4.2: Exemplo da uma especificação OCL ao método *type()* da classe *TriTyp*.

A especificações contidas no Código 4.2 ilustram as restrições de pós-condição para o método *type*(), restrições essas que dizem respeito ao valor que o atributo *trityp* deve possuir de acordo com o valor dos demais atributos da classe. São eles:

- 1: se o formato do triângulo for escaleno (todos os lados do triângulo de dimensões diferentes);
- 2: se o formato do triângulo for isósceles (dois lados do triângulo são de mesma dimensão);
- 3: se o formato do triângulo for equilátero (todos os lados do triângulo tem a mesma dimensão);
- 4: se as dimensões informadas não formam um triângulo (um dos lados do triângulo tem tamanho inferior ou igual a zero, ou a soma de dois dos lados do triângulo é superior a dimensão do terceiro).

4.2 Operadores de Mutação

Nessa seção são descritos os operadores de mutação utilizados para localizar defeitos presentes na especificação. Esses operadores foram propostos considerando os operadores para contratos propostos por Jiang *et al.* [JHS+05], e apresentados na Seção 3.4.

É importante considerar que os operadores de mutação são aplicados à cada condição de uma restrição.

Os operadores são classificados em quatro grupos conforme a Tabela 4.1. O primeiro grupo é responsável pela negação de uma condição presente na especificação. O segundo grupo busca por defeitos nas pré-condições, considera que uma condição muito forte foi estabelecida por engano para a pré-condição restringindo o número de chamadas ao método. Propõe que essa condição seja enfraquecida, trocando-se operadores relacionais, e permitindo que o método seja chamado um número maior de vezes. O terceiro grupo considera que uma condição muito fraca foi estabelecida como restrição de pós-condição. Os operadores de mutação do terceiro grupo operam fortalecendo uma condição da especificação, permitindo que o resultado pertença a um domínio mais limitado. O quarto grupo considera que determinada condição é desnecessária, ou seja, tanto em uma pré-condição como em uma pós-condição determinada validação deve ser ignorada. Nesse caso o operador de mutação substitui a condição por *TRUE*, deixando-a sempre válida.

Tabela 4.1: Descrição dos operadores de mutação

Operadores de Mutação	**Regras**		**Exemplo**	
	Original	Mutante	Original	Mutante
(CN) Condição Negada	(val)	$not\ (val)$	$(x == y)$	$!\ (x == y)$
(PE)Pré-Condição Enfraquecida	$==$ $<$ $>$	$>=, <=$ $<=$ $>=$	$(x == y)$ $(x < y)$ $(x > y)$	$(x >= y), (x <= y)$ $(x <= y)$ $(x >= y)$
(PF) Pós-Condição Fortalecida	$>=$ $<=$	$>, ==$ $<, ==$	$(x >= y)$ $(x <= y)$	$(x > y), (x == y)$ $(x < y), (x == y)$
(CD) Condição Desnecessária	(val)	$TRUE$	$(x > y)$	$TRUE$

4.3 Dado de teste

Um dado de teste para a abordagem é considerado um programa simples em OO que instancia um objeto da classe em teste, carrega os parâmetros necessários e executa o método em teste.

O Código 4.3 ilustra um exemplo em Java de um dado de teste para o método em teste *void type()* da classe *TriTyp*. Descrevendo o código tem-se a instânciação de um objeto da classe *TriTyp* para a variável *t1* no código *TriTyp t1 = new TriTyp()*, os valores dos parâmetros *i*, *j* e *k* são definidos respectivamente nos comandos *t1.setI(6)*, *t1.setJ(5)* e *t1.setK(9)*, e há uma chamada ao método em teste *type()* no código *t1.type()*.

```
public class TriTypDataTest01 {

    public static void main(String [] args) {
        TriTyp t1 = new TriTyp();
        t1.setI(6);
        t1.setJ(5);
        t1.setK(9);
        t1.type();
    }
}
```

Código 4.3: Exemplificando um dado de teste

4.4 Utilizando POA

POA permite o desenvolvimento de aspectos associados a uma determinada classe OO. Aspectos esses que são executados juntamente com a classe, realizando diversas operações com os mais diversos interesses com relação ao interesse da classe, chamados de interesses transversais.

A checagem das condições definidas em uma especificação original ou modificada podem ser vistas como interesses transversais aos interesses da classe. Por isso podem ser instrumentados usando POA. O primeiro passo para essa instrumentação é determinar em que ponto de execução tal checagem deve ser executada.

Os pontos de execução utilizando POA são também conhecidos como pontos de junção. Nos pontos de junção é possível definir um adendo a ser executado. Um adendo é um

bloco de código, que entre outras informações tem as palavras reservadas *after* ou *before* que determinam em que ponto de junção o adendo deve ser executado.

Para a instrumentação das especificações OCL, uma pré-condição deve ser executada antes da execução do método correspondente, já uma pós-condição deve ser executada justamente após a execução do método, correspondendo respectivamente ao adendo possuir *before* e *after* definidos no início de sua declaração.

O Código 4.4 ilustra o formato de um adendo que a abordagem considera para a instrumentação de uma pré-condição com o tipo de adendo *before* e de uma pós-condição com o tipo de adendo *after*.

```
before([lista de parâmetros]) :
      call ([lista de assinaturas]) && target([objeto da chamada]
                     && args([argumentos]))  {
   [bloco de código]
}

after([lista de parâmetros])
                 returning([retorno]) :
      call ([lista de assinaturas]) && target([objeto da chamada]
                     && args([argumentos]))  {
   [bloco de código]
}
```

Código 4.4: Formato dos adendos de pré e pós-condição

O Código 4.4 possui alguns parâmetros que significam:

- Lista de parâmetros: Lista de objetos e/ou tipos primitivos que serão utilizados pelo bloco de código do adendo. São exemplos: *double i, int x, String y* e *TriTyp t*;
- Lista de assinaturas: Lista de assinaturas de métodos que quando invocados o adendo deverá ser executado. São exemplos: *void type(), String sayHello()* e *double somar(int i, int j)*;
- Objeto da chamada: Instância (objeto) da classe que está sendo testada.
- Argumentos: Lista de objetos e/ou tipos primitivos que deseja-se obter acesso no momento da chamada do método, o qual também é um parâmetro do adendo.

- Retorno: Só é informado quando tipo do adendo é *after* e quando um dos métodos chamados possuir algum valor de retorno. Quando o retorno do método é *void* a palavra reservada *returning* não é necessária. São exemplos de retornos: *double res, String result e int n.*

- Bloco de código: Nele são incluídos as checagens a serem realizadas conforme validações definidas pelas condições da especificação OCL. No Código 4.5 exemplifica-se como o parâmetro "bloco de código" é instanciado.

Exemplificando a utilização de POA na abordagem, mais especificamente utilizando a linguagem AspectJ, o Código 4.5 ilustra a instrumentação de um adendo de uma pós-condição do método *void type()* da classe *TriTyp*.

```
    after(TriTyp t) :
         call (void TriTyp.type()) && target(t)
    {
         if (((t.i <= 0) || (t.j <= 0) ||
              (t.k <= 0) || (t.k > (t.i + t.j)) ||
              (t.i > (t.j + t.k)) ||
              (t.j > (t.i + t.k)))) {
              // satisfaz restrição 1
         } else
         if ((t.i == t.j) & (t.i == t.k)) {
              // satisfaz restrição 2
         } else
         if ((t.i == t.j) ||
             (t.i == t.k) ||
             (t.j == t.k)) {
              // satisfaz restrição 3
         } else {
              // satisfaz restrição 4
         }
    }
}
```

Código 4.5: Instrumentação de um adendo de pós-condição

Com base em todas as informações da atual seção e das anteriores desse capítulo, uma visão geral da abordagem proposta foi apresentada, nas próximas duas seções um exemplo de cada um dos dois usos é descrito.

4.5 Uso 1: Geração de dados de teste a partir da especificação

A não implementação de determinada funcionalidade definida na especificação de um método de uma classe produz uma das limitações inerentes às técnicas estrutural e baseada em defeitos. Se uma determinada funcionalidade não foi implementada, não existirá nenhum caminho a ser percorrido que lhe corresponda. Essa situação é conhecida como *caminho ausente*.

A existência de caminhos ausentes, quando o teste é realizado somente sobre a implementação de um programa, é a principal motivação da validação da classe considerando sua especificação. Os defeitos causados por caminhos ausentes podem nunca ser identificados pelos testes gerados somente a partir da implementação. Os casos de teste gerados, por exemplo, pela técnica estrutural podem chegar a cobrir inteiramente os critérios todos-os-nós, todas-as-arestas e até mesmo todos os caminhos do programa e mesmo assim não identificar um caminho ausente.

A utilização da especificação da classe como base para geração dos casos de teste auxilia a identificar esse tipo de defeito.

O uso da abordagem para geração de dados de teste a partir da especificação resume-se na seguinte descrição: a partir de uma especificação OCL descrita conforme gramática sugerida pela abordagem, gera-se um aspecto, com pré-condições e pós-condições que apontam as restrições de cada método da classe, conforme a especificação. Com base em um conjunto de dados de entrada de teste pré-definido pelo usuário, os aspectos de cada classe são executados, gravando resultados pertinentes da sua execução. Os resultados na execução de cada dado de entrada sobre os aspectos permitem avaliar se a classe está de acordo com a sua especificação, e ainda, caso a classe não esteja de acordo, qual a restrição que a classe não atende.

Exemplificando o uso dessa abordagem considera-se a classe $TriTyp$ como a classe em teste.

Supondo que o método $type()$ da $TriTyp.java$ foi implementado conforme a Figura 4.2, observa-se que por descuido ou desconhecimento da especificação, a classe não foi corretamente implementada, deixando sem tratamento a situação que $(trityp = 4)$ na qual os

lados informados não formam um triângulo. Isso pode ser considerado como um caminho ausente na implementação com relação a sua especificação.

```
public class TriTyp {

  private int i, j, k;
  public int trityp;

  public void setI(int x) {i = x;}
  public void setJ(int x) {j = x;}
  public void setK(int x) {k = x;}

  public void type() {
    if (i==j) {
        trityp=trityp+1;
    }
    if (i==k) {
        trityp=trityp+2;
    }
    if (j==k) {
        trityp=trityp+3;
    }

    if (trityp == 0) {
        trityp=1; //escaleno
    } else if (trityp>3) {
      trityp=3;  // equilátero
    } else {
      trityp=2;  // isosceles
    }
  }
}
```

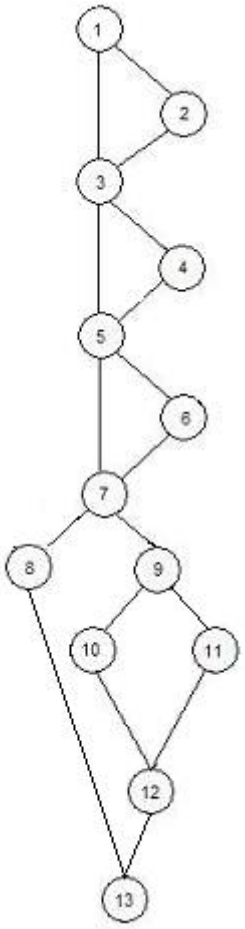


Figura 4.2: *TriTyp.java* implementado com defeito de caminho ausente e Grafo de Fluxo de Controle do método *type*().

Nessa situação, a geração de dados a partir da implementação pode não identificar o defeito, por exemplo a Figura 4.2 ilustra o código com defeito e o grafo de fluxo de controle referente ao método *type*(), observando pode-se perceber que todos os caminhos executáveis desse grafo podem ser cobertos sem que o defeito seja encontrado, por exemplo executando os seguintes dados de teste ($i = 3$; $j = 3$; $k = 1$), ($i = 1$; $j = 1$; $k = 1$), ($i = 3$; $j = 5$; $k = 4$), ($i = 4$; $j = 5$; $k = 5$) e ($i = 3$; $j = 4$; $k = 3$). Se a classe for instrumentada com o aspecto do Código 4.5, utilizando-se o mesmo conjunto de dados de teste acima, apenas as restrições 2, 3 e 4 são satisfeitas. O usuário é obrigado a gerar um novo dado de teste para satisfazer a restrição 1, tal como o dado ($i = 7$; $j = 1$; $k = 0$) que irá revelar o defeito.

4.6 Uso 2: Validação da especificação e da implementação por meio da análise de especificações mutantes

A motivação para o uso da abordagem para validação da especificação e da implementação de uma classe por meio de análise de especificações mutantes é a premissa de que as especificações de uma classe não estão livres de defeitos. Além disso, o teste tradicional de mutação baseado nas execuções do programa tem alto custo.

A utilização de mutantes para validar a especificação de uma classe faz com que dados de teste possam ser testados pensando não somente em defeitos introduzidos na implementação por conta de enganos dos programadores, mas também para descrever defeitos presentes na especificação.

Cada restrição definida na especificação como pré-condição ou pós-condição de um método de uma classe está sujeita a conter um defeito que será descrito por uma mutação. A execução de um determinado conjunto de dados de teste considerando a especificação original e uma especificicação que sofreu mutação retorna valores que devem ser avaliados na execução do programa, afim de definir qual a especificação está correta para a classe.

A utilização de aspectos junto com mutação parece ser bastante promissor. Como na utilização de aspectos, não existe a necessidade de se alterar a classe base, diversos aspectos podem ser gerados conforme o número de mutantes, e o mais interessante é que todos os aspectos podem ser executados e todas as restrições testadas em uma única execução da classe, reduzindo custos.

O uso da abordagem para validação da especificação e da implementação por meio da análise de especificações que sofreram mutação resume-se na seguinte descrição, a partir de uma especificação OCL descrita conforme gramática sugerida pela abordagem, efetuam-se mutações conforme os operadores de mutação pré-definidos, geram-se especificações modificadas para a classe em questão. Para cada especificação modificada gera-se um aspecto com as pré-condições e pós-condições impondo as restrições apontadas a cada método. A partir de um conjunto de dados de teste definidos pelo usuário, os aspectos de cada especificação modificada são executados, gravando os resultados obtidos sobre cada

restrição instrumentada. Os resultados obtidos pela execução dos dados de teste em cada aspecto de uma determinada especificação modificada, servem para avaliar se diferenças foram encontradas com relação aos resultados da execução do aspecto da especificação original, o que pode implicar em um defeito na especificação da classe. Caso existam diferenças entre os resultados fica a cargo do testador avaliar se a especificação que sofreu mutação não é a correta para o programa revelando assim um defeito na especificação original. Caso não existam diferenças entre os resultados da execução dos dados de teste na especificação original e modificada, fica a cargo do testador avaliar se trata-se de uma especificação equivalente.

Exemplificando, considera-se o Código 4.6 como parte do código OCL de especificação da classe *TriTyp* referente ao método *type*(), com defeito. Pode-se reparar que incorretamente foi definido o operador condicional >= ao invés de = na validação ($self.j >= self.k$)) contida na sexta linha da especificação da pós-condição.

```
context TriTyp::type():void
  pre:
  post: if ( (self.i<=0) or (self.j<=0) or
        (self.k<=0) or (self.i>=(self.j+self.k))
        or (self.j>=(self.i+self.k)) or
        (self.k>=(self.i+self.j)) then
        self.trityp=4 else if ((self.i=self.j)
        and (self.j>=self.k)) then self.trityp=3  //=> com defeito
        else if ((self.i==self.j) or
        (self.i=self.k) or
        (self.j=self.k)) then self.trityp=2 else
        self.trityp=1 endif endif endif
```

Código 4.6: Código OCL de especificação da pós-condição do método *type*() com defeito.

Supondo que o método *type*() foi implementado conforme a especificação incorreta, o defeito estará presente na implementação. Os dados de teste ($i = 3$; $j = 3$; $k = 1$), ($i = 1$; $j = 1$; $k = 1$), ($i = 3$; $j = 5$; $k = 4$) e ($i = 7$; $j = 1$; $k = 0$) que satisfazem o Uso 1 não revelariam o defeito.

Utilizando a mutação da especificação, nesse exemplo especificamente através do operador PF, diversas especificações modificadas seriam geradas e implementadas como aspectos, entre elas uma em que a condição com defeito seria subtituída por ($self.j > self.k$). Essa especificação modificada, exigiria um novo dado de teste, igual a ($i = 3$;

$j = 4$; $k = 3$), para cobertura completa das restrições, revelando assim o defeito.

4.7 Considerações Finais

Esse capítulo apresentou e exemplificou a abordagem proposta neste trabalho. A abordagem tem como base a utilização de um subconjunto de expressões da especificação OCL. Subconjunto esse que é limitado em considerar as restrições condicionais de pré e pós-condições especificadas, alguns tipos e operações.

O uso da abordagem para geração de dados a partir da especificação tem como vantagem realizar um teste funcional que considere o total de restrições de uma especificação, como o total de elementos requeridos a serem cobertos auxiliando a identificar defeitos, principalmente de caminhos ausentes que não seriam identificados no teste sem a especificação.

O uso da abordagem para validação da especificação e da implementação por meio da análise de especificações que sofreram mutações permite a identificação de defeitos presentes na especificação. É possível também considerar as restrições que sofreram mutação como elementos requeridos a serem considerados em uma avaliação de cobertura. O uso de aspectos reduz custo porque reduz o número de execuções do programa, geralmente adotadas pelo teste de mutação tradicional.

O desenvolvimento de uma ferramenta que implemente a abordagem e permita os usos aqui descritos é assunto do próximo capítulo.

CAPÍTULO 5

ASPECTOS DE IMPLEMENTAÇÃO

Para instrumentar a abordagem estudada foi implementada a ferramenta **MuSA** (teste de **Mu**tação baseado em E**S**pecificações OCL e **A**spectos). O desenvolvimento da ferramenta permite a utilização e avaliação prática da abordagem.

A ferramenta MuSA tem algumas características distintas que formalizam e limitam a aplicação das abordagens estudadas, características estas que serão o assunto da continuidade desse capítulo, que descreve seus principais aspectos de implementação

Apesar da abordagem estudada possuir fundamentação genérica e possibilidade de aplicação em diversas linguagens de programação que implementem os conceitos básicos da orientação a objetos e aspectos, a ferramenta propõe a aplicação da abordagem somente sobre classes implementadas em Java.

5.1 Formato da especificação esperada para uma classe

As especificações das classes aceitas pela ferramenta são no formato XML. Foi definido um esquema XML (MuSATestEsquema.xsd) com regras para o formato da especificação XML ser válido de acordo com as necessidades da ferramenta.

Especificações definidas em OCL ou outra linguagem de contratos, devem ser convertidas no formato XML definido pelo esquema para serem utilizadas como base pela ferramenta. O conteúdo do esquema XML utilizado é apresentado no Apêndice B. As principais informações requeridas na especificação são referentes às restrições de cada uma das pré e pós-condições que cada método possui, além do detalhamento dos tipos dos dados de cada um dos elementos que fazem parte das restrições.

5.2 Modo de operação da ferramenta

Atualmente a ferramenta MuSA trabalha com projetos de teste.

A cada projeto de teste devem ser informados a classe base de teste; a especificação da classe a ser considerada; os dados de teste que devem ser executados e operadores de mutação a serem utilizados. As telas de configurações iniciais do projeto e de seleção de operadores de mutação são ilustradas nas Figuras 5.1 e 5.2 respectivamente.

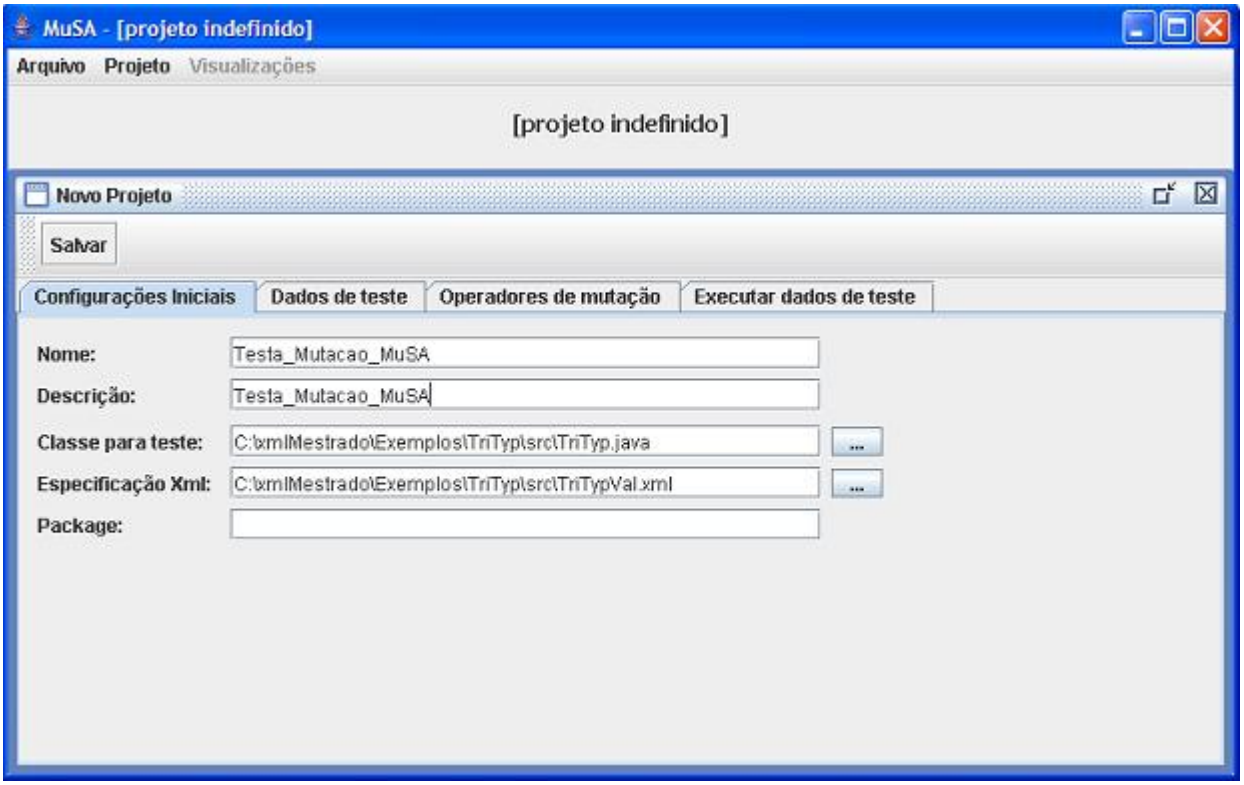


Figura 5.1: MuSA - novo projeto - configurações iniciais.

É importante considerar que outras classes podem fazer parte do programa que está sendo testado, porém somente a classe informada como base do projeto é que vai ser instrumentada com os aspectos para ser avaliada.

5.3 Instrumentação do teste com aspectos

No capítulo anterior foi destacado como um ponto importante, a possibilidade de instrumentação dos testes sem alteração do código fonte da classe de teste.

Uma das formas dessa instrumentação sobre classes é com a utilização de aspectos

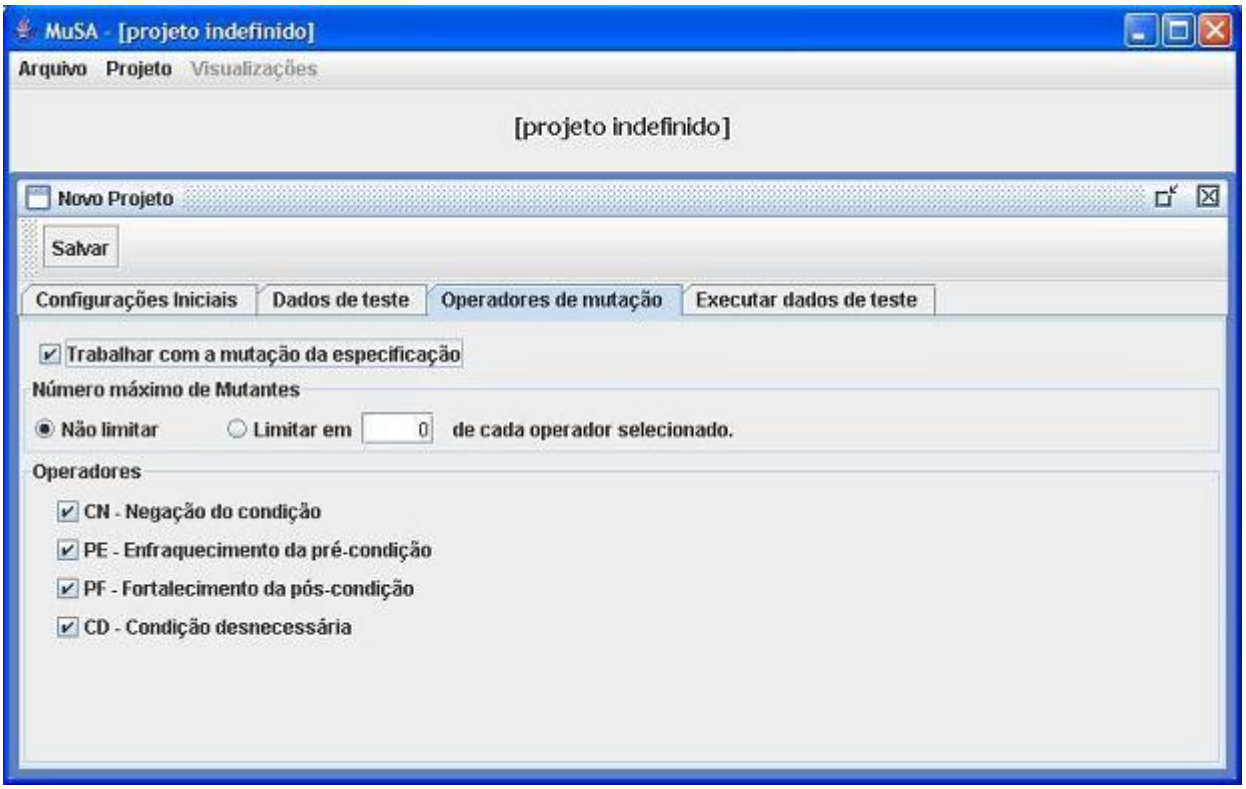


Figura 5.2: MuSA - novo projeto - seleção de operadores de mutação.

da POA. A ferramenta MuSA visando a instrumentar as classes em Java trabalha com a instrumentação da classe utilizando a linguagem de aspectos AspectJ.

O processo de geração de aspectos pode ser descrito como a interpretação das restrições de pré-condições e pós-condições da especificação, e geração de código em AspectJ referente à especificação, conforme os pontos de junção discutidos no capítulo anterior.

5.4 Dados de teste

Os dados de teste são uma das entradas para a ferramenta. Tratam-se de pequenos programas em Java que, em suas diversas chamadas instanciam um objeto da classe em teste e também fazem chamadas a seus métodos. A Figura 5.3 ilustra a entrada de dados de teste na ferramenta.

Mais informações sobre um dado de teste válido à ferramenta, assim como um exemplo de um dado de teste válido, foram apreesentados na Seção 4.3.

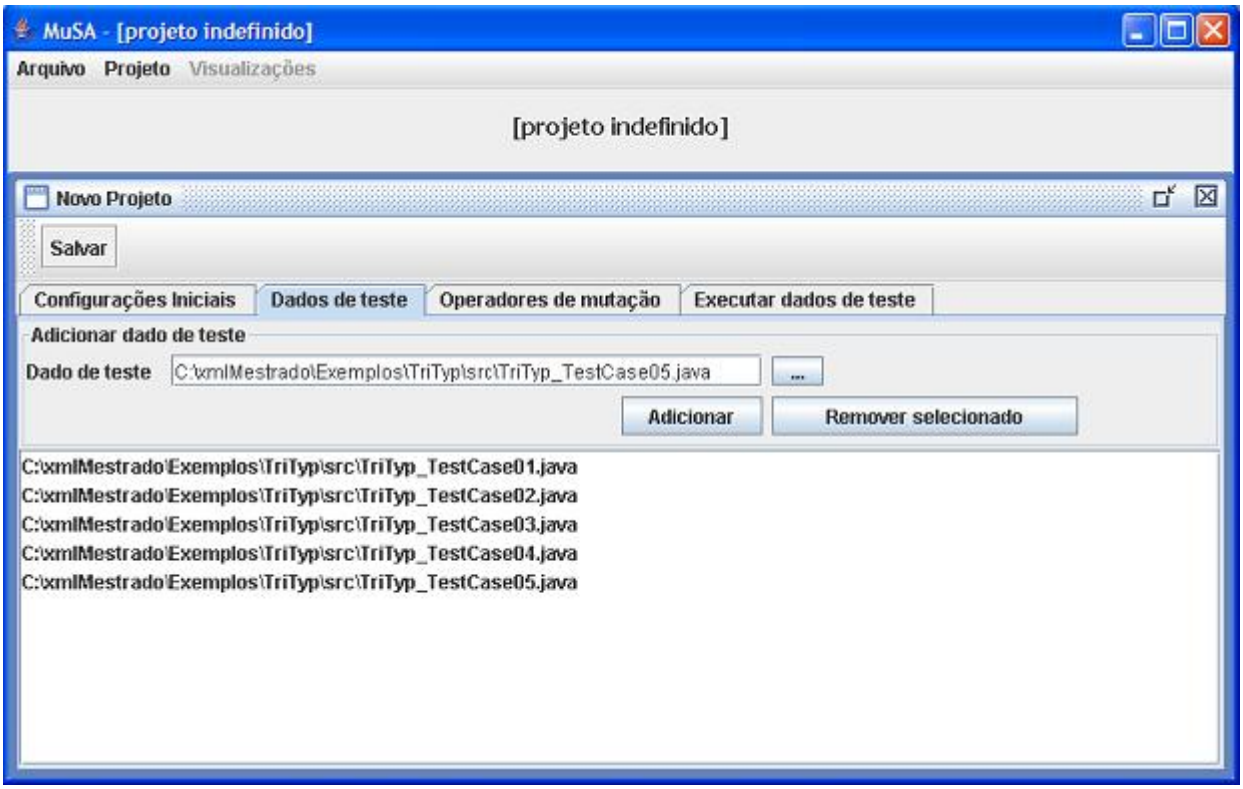


Figura 5.3: MuSA - novo projeto - seleção de dados de teste.

5.5 Operadores de mutação disponíveis

Para a abordagem de validação da especificação e da implementação por meio da análise de especificações modificadas, o conjunto de operadores de mutação descritos na Seção 4.2 foi utilizado.

A Figura 5.4 ilustra o diagrama de classes utilizadas para a implementação dos operadores de mutação na ferramanta. Toda implementação parte da classe *Mutacao* que tem as funções básicas do trabalho com os operadores de mutação, a classe *Operador* é associada a classe *Mutacao* e tem os métodos de geração do código modificado. As classes *CN*, *PE*, *PF* e *CD* são especializações da classe *Mutacao*, elas contêm a sobreposição dos métodos da classe *Mutacao* implementando cada uma as características do operador de mutação referido.

Antes do processo de geração de mutantes, o usuário pode escolher quais são os operadores que ele deseja aplicar. Isso permite ao usuário uma flexibilidade quanto ao tipo de mutação que ele deseja que o teste aborde.

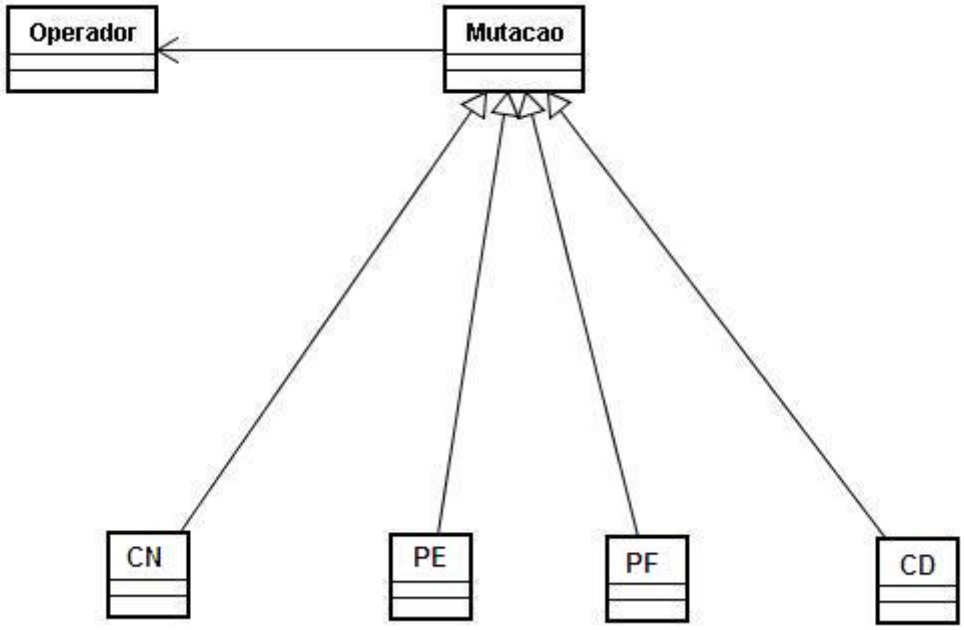


Figura 5.4: Diagrama de classes ilustrando a implementação dos operadores de mutação pela ferramenta.

5.6 Processos internos da ferramenta

Após definidas as entradas em um novo projeto para a ferramenta MuSA, fica por conta da ferramenta o processamento das informações para disponibilização dos resultados.

Ordenadamente podem-se considerar os seguintes processos internos como principais que a ferramenta executa:

1. Validação da especificação da classe: A especificação da classe informada pelo usuário passa por um processo de validação com relação ao esquema XSD fornecido pela ferramenta. Essa validação é decisiva no que diz respeito à continuidade do processo, caso o arquivo de especificação tenha alguma não conformidade com o esquema XSD sugerido, o processo é abortado;

2. Definição de especificações modificadas: Caso selecionado pelo usuário a utilização de mutação da especificação, esse processo interno é executado. O processo trata de gerar em memória especificações modificadas confome os operadores de mutação préviamente selecionados. A quantidade máxima de especificações modificadas a ser gerada por cada operador também é uma entrada que pode ser fornecida pelo

usuário.

3. Instrumentação da(s) especificação(ões) em aspecto(s): Conforme a abordagem selecionada para o trabalho pela ferramenta, nesse processo tem-se somente a especificação da classe informada, ou junto com a especificação da classe também todas as especificações modificadas que foram geradas pela ferramenta. Esse proesso de instrumentação gera um aspecto para cada especificação carregada na ferramenta, seja ela original ou modificada. Os aspectos gerados são guardados em arquivos na mesma pasta onde encontram-se a classe base do teste.

4. Geração da classe *MainClass*: A ferramenta MuSA gera uma classe que em seu código tem instrumentado que todos os aspectos gerados devem ser chamados junto a execução de cada dado de teste. Nessa classe também é definido em que arquivo vão ser salvas as informações referentes ao teste realizado pela instrumentação do aspecto.

5. Compilação de todos os códigos influentes: Desde a classe base, como os dados de teste, os aspectos gerados, como a classe *MainClass*, passam por um processo de compilação. Essa compilação vai gerar os *bytecodes* Java que são executados na sequência pela ferramenta.

6. Execução da classe *MainClass*: Trata-se da execução dos dados de teste sobre a classe base com os aspectos gerados. Como na geração da classe *MainClass* foram interligados todos os aspectos para serem executados a cada chamada de cada dado de teste, e definido para cada aspecto um arquivo de saída para os resultados obtidos, a simples execução da classe *MainClass* faz com que o todos os dados de teste sejam executados sobre a classe base, e como a classe base é executada junto com seus aspectos, os aspectos também são executados gravando as informações dos resultados dos testes exercitados nos arquivos de saída definidos. Entre as informações que são gravadas destacam-se detalhes sobre quais métodos em teste foram executados e se os mesmos foram executados com sucesso ou não. Esses resultados são detalhados na próxima seção.

Todo o processo interno apresentado acima é iniciado pelo usuário ao clicar no botão *Compilar* da tela de processamento da ferramenta como ilustrado na Figura 5.5.

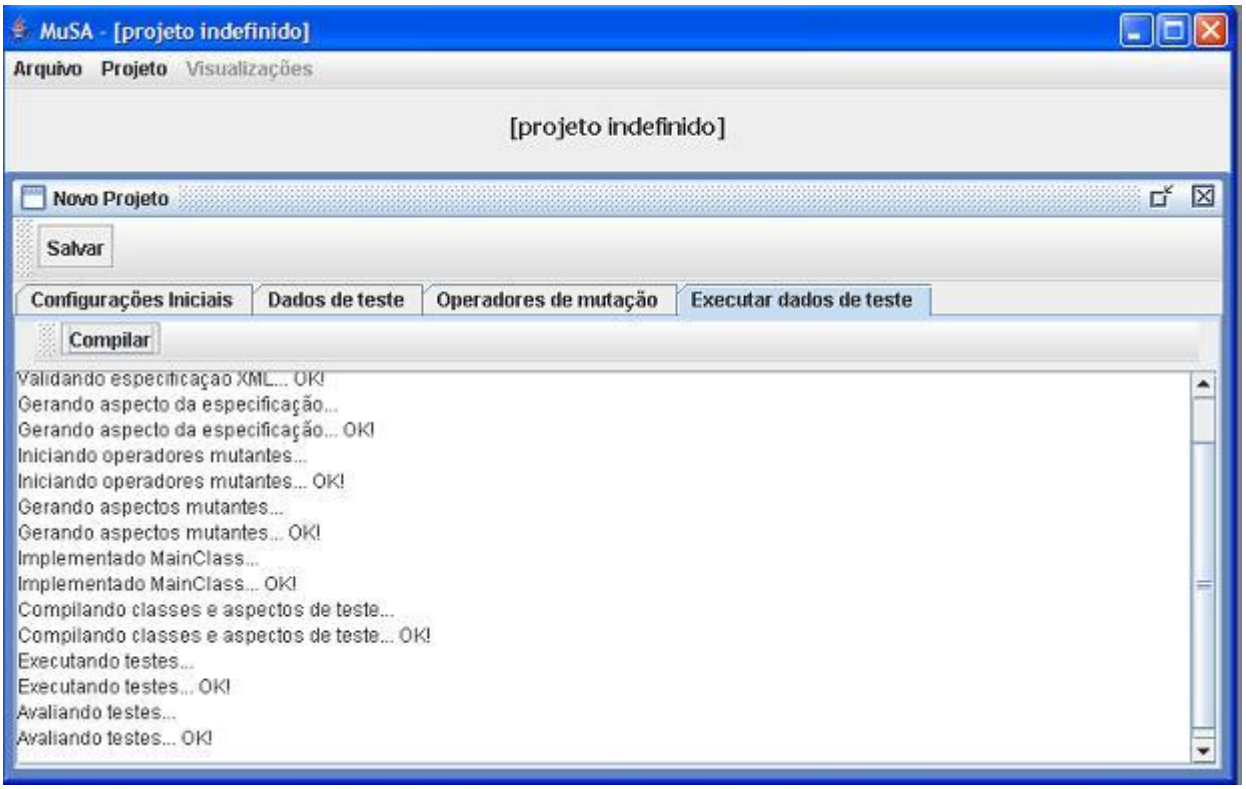


Figura 5.5: MuSA - novo projeto - tela de processamento.

5.7 Resultados da ferramenta

A MuSA na execução de seus processos internos grava diversas informações sobre a execução de cada dado de teste sobre cada aspecto gerado pela ferramenta. Para melhor ilustração desses resultados ao usuário, a ferramenta disponibiliza três visualizações principais: "Avaliação de mutantes", "Cobertura das restrições pelos dados de teste" e "Cobertura das restrições pelos dados de teste para mutantes".

A visualização "Avaliação de mutantes" apresenta os resultados em três grupos de informações que são:

- Detalhes dos dados de teste: Apresenta por dado de teste informação da execução feita em cada mutante. As informações de cada mutante são: tipo do operador de mutação utilizado, qual a pré-condição ou pós-condição que sofreu mutação, qual a

restrição original que sofreu a mutação, como ficou a restrição que sofreu mutação e ainda qual o resultado da execução do dado de teste, se o mutante foi morto, ficou vivo ou não foi exercitado, resultado relacionado com a satisfação ou não da restrição.

- Resumo dos dados de teste: Apresenta um resumo das informações do teste realizado por dado de teste. As informações por dado de teste apresentadas são: total de mutantes do projeto, total de mutantes mortos, total de mutantes vivos e total de mutantes exercitados, e ainda percentuais relativos a mortos e vivos sobre os mutantes totais e também sobre os que foram exercitados.

- Relação de mortos, vivos e não exercitados: Apresenta por dado de teste a listagem de cada mutante morto, vivo e não exercitado.

Ambas visualizações "Cobertura das restrições pelos dados de teste" referente ao Uso 1, e "Cobertura das restrições pelos dados de teste para mutantes" referente ao Uso 2, indicam quais restrições cada dado de teste cobre, quais os dados de teste que quando executados cobrem alguma restrição ainda não coberta, e ainda, a quantidade de restrições cobertas e a quantidade de restrições totais. A partir delas pode-se avaliar se determinada especificação teve suas restrições todas cobertas pelo conjunto de dados de teste

As telas de resultado da ferramenta são ilustradas no Apêndice C.

5.8 Considerações Finais

Esse capítulo apresentou a ferramenta MuSA. Trata-se de uma ferramenta que, com suas limitações, implementa a abordagem proposta e permite sua utilização e avaliação na prática, sem a ferramenta isso seria impossível. A ferramenta será base de instrumentação para os experimentos que serão apresentados no próximo capítulo.

CAPÍTULO 6

EXPERIMENTO

Visando à aplicação e validação da abordagem proposta e de seus dois usos, com base na ferramenta desenvolvida, o capítulo atual documenta o experimento realizado focando nos seus objetivos, metodologia utilizada e resultados encontrados.

6.1 Objetivos

O atual experimento visa a avaliar a abordagem proposta vendo sua aplicabilidade e custo em termos de número de dados de teste necessários. Para isso serão considerados os dois usos descritos no Capítulo 4. Além disso a abordagem será comparada com os critérios estruturais implementados pela ferramenta JaBUTi.

6.2 Programas

Para o experimento foram utilizados os seguintes programas (classes Java):

- *TriTyp.java* ilustrado no Código 4.2, com a instrumentação do método *type*();
- *Ordena.java*, com a instrumentação do método *ordena*();
- *Data.java*, com a instrumentação do método $diasNoMes()$;
- e *Fourballs.java* com a instrumentação do método $relativeWeight()$.

Alguns destes programas foram utilizados por Clarke e Richardson [CR83], mas o código de todos eles foi implementado especialmente para esse trabalho.

Os métodos escolhidos para a instrumentação são os que apresentam maior complexidade e cuja funcionaliade está especificada em termos de pré e pós-condições.

6.3 Metodologia

Conforme o objetivo proposto, os dois usos da abordagem e os critérios da ferramenta JaBUTi serão avaliados e comparados, são eles: o uso "Geração de dados de teste a partir da especificação" nomeado para o experimento como Uso 1, o uso "Validação da especificação e da implementação por meio da análise de especificações mutantes" nomeado para o experimento como Uso 2, ambos implementados pela ferramenta MuSA, e os critérios estruturais implementados pela ferramenta JaBUTi: Todos-Nós, Todas-Arestas, Todos-Usos e Todos-Pot-Usos descritos na Seção 3.3.1.

A avaliação foi realizada considerando uma metodologia que é definida em três etapas principais. Nessa metodologia os usos para a abordagem proposta são também vistos como critérios de teste, pois satisfazer as pré e pós-condições podem ser vistos como elementos requeridos em cada uso. São as três etapas principais da metodologia:

1. Avaliação das abordagens com um conjunto de dados de teste inicial: Para cada programa foi definido um conjunto de dados de teste inicial nomenado de Ti. Ti é formado por valores aleatoriamente gerados por um programa auxiliar, o sorteio dos valores foi sempre considerando uma faixa de limite equivalente a valores válidos e também inválidos para as entradas dos programas. Os dados de teste são então igualmente definidos e na sequência executados para cada uma das abordagens. Os resultados de cada uma das abordagens são armazenados. Destacam-se nos resultados os elementos (ou restrições) que foram realmente cobertos e não cobertos, e ainda quais os dados de teste que foram realmente utilizados para obter a maior cobertura. Esses dados passam a fazer parte do conjunto de dados de teste adequado a cada critério, nomeado de Ta.

2. Definição de um conjunto de dados de teste mais adequado: Com base nos resultados da execução do conjunto de dados de teste inicial, é definido individualmente para cada uma das abordagens um novo conjunto de dados de teste. Desse novo conjunto os dados que quando executados contribuírem para uma melhora na cobertura no teste realizado sobre o programa são armazenados no conjunto de dados de teste Ta.

Novos dados de teste devem ser definidos manualmente para cada programa e cada critério até chegar ao ponto em que todos os elementos ou tenham sido cobertos, ou identificados como não executáveis. Com isso no término dessa etapa é possível armazenar os valores de melhor cobertura de cada programa, e identificar um Ta para cada critério em cada programa. Lembrando que o Ta contém a informação dos dados de teste utilizados para conseguir a melhor cobertura.

3. Submissão do conjunto Ta adequado a um critério para avaliação de cobertura de outro critério: Nessa etapa o conjunto Ta para o Uso 1 foi avaliado segundo o Uso 2 e segundo os critérios da JaBUTi, assim como o conjunto Ta do Uso 2 foi avaliado segundo o Uso 1 e os critérios da JaBUTi e, os conjuntos Ta dos critérios da JaBUTi foram avaliados segundo os Usos 1 e 2. Os resultados de tais execuções também foram armazenados. Entre os resultados obtidos, os principais são os elementos cobertos ou não. Tal etapa é importante para avaliar a relação de inclusão entre os critérios implementados; ou seja, ver se os dados de teste utilizados por um critério satisfazem ou não um outro.

As etapas acima citadas foram executadas para cada programa. Os programas $TriTyp$, $Ordena$ e $Fourballs$ tiveram o conjunto inicial (Ti) com um total de 20 dados de teste, já o programa $Data$ por se tratar de um programa com entradas de dados mais restrita teve um conjunto Ti com somente 10 dados de teste.

6.4 Resultados

Os resultados do experimento realizado com os programas apresentados na Seção 6.2, confome a metodologia apresentada na Seção 6.3 foram tabelados e são apresentados nessa seção.

Para facilitar a nomenclatura dos termos das abordagens foi considerado de mesmo valor funcional o termo "restrição" e o termo "elemento requerido" definido pela JaBUTi, sendo que no restante dessa seção será utilizado somente o termo "elemento requerido".

As tabelas de resultado apresentam a sigla $E/C/T$, sendo que E corresponde ao

número total de elementos requeridos, C correponde ao número de elementos cobertos e T é o número de dados de teste utilizados para a cobertura.

A Tabela 6.1 ilustra os valores dos dados de teste iniciais utilizados para cada programa.

Tabela 6.1: Conjuntos de dados de teste Ti executados para cada programa.

Nro do dado de teste	TriTyp e Ordena Ti			Data Ti			Fourballs Ti				
	i	j	k	dia	mes	ano	mA	mB	mC	mD	cual
1	6	5	9	5	7	2000	8	-3	6	1	-7
2	7	3	5	12	8	1977	8	-6	2	4	-4
3	9	7	6	20	6	1983	9	6	-7	8	5
4	1	3	7	14	6	1955	7	8	8	2	-2
5	8	5	9	7	1	1959	9	-4	-9	6	0
6	7	6	7	3	7	1987	0	3	0	3	6
7	7	1	1	24	2	1994	-5	-5	-5	-8	2
8	1	3	5	1	12	1988	1	-9	-8	4	-6
9	5	5	8	7	11	2006	0	4	-2	-1	-4
10	6	2	1	15	4	2009	6	8	1	2	-4
11	3	6	2				-8	1	-8	3	7
12	8	3	7				-6	-2	-9	7	7
13	3	1	8				9	-9	-2	-7	-7
14	7	5	3				4	1	-3	-6	3
15	1	2	8				5	-8	-5	0	-6
16	5	4	4				2	2	-8	6	2
17	9	2	1				5	9	-6	-6	7
18	4	1	4				5	8	9	4	-8
19	5	5	7				-1	6	6	-6	-6
20	1	8	5				-7	4	-4	-3	-7

A partir da execução dos testes com o conjunto Ti para cada programa são obtidos os resultados de cobertura ilustrados na Tabela 6.2.

Tabela 6.2: Resultados da execução do coonjunto Ti para cada programa.

Critérios/Programas	TriTyp	Ordena	Data	Fourballs	TOTAIS
	E/C/T	E/C/T	E/C/T	E/C/T	E/C/T
Uso 1	4/3/3	6/6/6	4/3/3	5/3/2	19/15/14
Uso 2 - CN	48/30/8	36/36/6	72/49/5	90/49/4	246/164/23
Uso 2 - PE	0/0/0	0/0/0	0/0/0	20/12/4	20/12/4
Uso 2 - PF	24/18/3	0/0/0	0/0/0	0/0/0	24/18/3
Uso 2 - CD	48/26/4	36/36/6	72/53/3	90/49/4	246/164/17
Uso 2 - Todos	120/74/8	72/72/6	144/102/5	200/110/4	536/358/23
JaBUTi - All-Nodes-ei	12/11/4	17/16/6	6/5/3	8/7/3	43/39/16
JaBUTi - All-Edges-ei	16/14/4	26/20/7	5/4/3	10/8/3	57/46/17
JaBUTi - All-Uses-ei	40/32/4	90/90/7	10/8/3	33/25/3	173/155/17
JaBUTi - All-Pot-Uses-ei	50/29/4	61/53/7	9/7/3	48/39/3	168/128/17
TOTAIS	242/237/42	344/329/51	322/231/28	504/302/30	1412/1099/151

Observa-se na Tabela 6.2 que alguns operadores (PE e PF) não se aplicam a todas às restrições presentes na especificação, isso ora por não ser possível conseguir enfraquecer a pré-condição conforme as restrições elaboradas na especificação (PE), ou pós-condição conforme as restrições elaboradas na especificação (PF).

Sobre o conjunto Ti de cada programa, os dados de teste ilustrados na Tabela 6.3 são os que cobrem os elementos requeridos em cada uma das abordagens do experimento para cada programa correspondente, e passam a fazer parte do conjunto Ta do programa.

Tabela 6.3: Dados de teste do conjunto Ta que cobrem restrições em cada uma dos critérios no experimento sobre cada um dos programas.

Critério	**TriTyp**	**Ordena**	**Data**,	**Fourballs**
Uso 1	1, 4, 6	1, 2, 3, 4, 11, 20	1, 3, 7	7, 14
Uso 2	1, 4, 6, 7, 9, 10, 11, 16	1, 2, 3, 4, 11, 20	1, 2, 3, 7, 9	1, 7, 14, 3
JaBUTi	1, 6, 7, 18	1, 2, 3, 4, 9, 11, 20	1, 3, 7	1, 7, 14

Em busca de um conjunto Ta com maior cobertura dos elementos requeridos para cada um dos programas, em um processo não automatizado, novos dados de teste foram adicionados. A exceção foi ao programa *Ordena*.

Os dados de teste adicionais aos conjuntos Ta de cada programa em cada abordagem, são apresentados na Tabela 6.4.

Tabela 6.4: Conjuntos de dados de teste adicionados no conjunto Ta de cada programa para uma melhor cobertura aos elementos requeridos em cada critério.

Critério	**Nro do Dado de Teste**	**TriTyp Ti**			**Data Ti**			**Fourballs Ti**				
		i	**j**	**k**	**dia**	**mes**	**ano**	**mA**	**mB**	**mC**	**mD**	**cual**
Uso 1	11	-	-	-	15	2	2008	-	-	-	-	-
	21	7	7	7	-	-	-	3	5	7	8	1
	22	-	-	-	-	-	-	-3	2	-9	3	4
Uso 2	11	-	-	-	15	2	2008	-	-	-	-	-
	21	7	7	7	-	-	-	3	5	7	8	1
	22	3	0	3	-	-	-	-3	2	-9	3	4
	23	2	2	0	-	-	-	0	3	0	3	6
	24	-	-	-	-	-	-	0	3	0	3	6
JaBUTi	11	-	-	-	15	2	2008	-	-	-	-	-
	21	7	7	7	-	-	-	3	5	7	8	1
	22	-	-	-	-	-	-	-3	2	-9	3	4

A partir do novo conjunto de dados de teste Ta, novamente foram executados os testes sobre cada um dos programas para os critérios. E os resultados obtidos nessa nova execução são ilustrados na Tabela 6.5.

Nos resultados da Tabela 6.5 percebe-se elementos requeridos ainda não cobertos. Esses elementos requeridos foram avaliados individualmente, nessa avaliação definidos como não executáveis.

Como última parte do experimento para cada programa, o conjunto Ta de cada abordagem foi utilizado para avaliação de cobertura nas outras abordagens. Dessa forma, a Tabela 6.6 ilusta os resultados da execução dos testes com o conjunto Ta do Uso 1 sobre

Tabela 6.5: Resultados da execução do conjunto de Ta para cada programa em cada critério.

Critérios/Programas	TriTyp	Ordena	Data	Fourballs	TOTAIS
	E/C/T	E/C/T	E/C/T	E/C/T	E/C/T
Uso 1	4/4/4	6/6/6	4/4/4	5/5/4	19/19/18
Uso 2 - CN	48/37/11	36/36/6	72/64/6	90/77/6	246/214/29
Uso 2 - PE	0/0/0	0/0/0	0/0/0	20/20/4	20/20/4
Uso 2 - PF	24/24/4	0/0/0	0/0/0	0/0/0	24/24/4
Uso 2 - CD	48/30/5	36/36/6	72/71/4	90/90/4	246/227/19
Uso 2 - Todos	120/91/11	72/72/6	144/135/6	200/187/8	536/485/31
JaBUTi - All-Nodes-ei	12/12/5	17/16/6	6/6/4	8/8/4	43/42/19
JaBUTi - All-Edges-ei	16/16/5	26/20/7	5/5/4	10/10/4	57/51/20
JaBUTi - All-Uses-ei	40/35/5	90/90/7	10/10/4	33/33/4	173/168/20
JaBUTi - All-Pot-Uses-ei	62/50/5	61/53/7	9/9/4	48/48/4	180/160/20
TOTAIS	374/299/55	344/329/51	322/304/36	504/478/42	1544/1410/184

o Uso 2 e os critérios da JaBUTi respectivamente, assim como, a Tabela 6.7 ilusta os resultados da execução dos testes com o conjunto Ta do Uso 2 sobre o Uso 1 e os critérios da JaBUTi, e a Tabela 6.8 ilusta os resultados da execução dos testes com o conjunto Ta da abordagem JaBUTi sobre o Uso 1 e o Uso 2.

Tabela 6.6: Resultados da avaliação do conjunto Ta do Uso 1 com os outros critérios.

Critérios/Programas	TriTyp	Ordena	Data	Fourballs	TOTAIS
	E/C	E/C	E/C	E/C	E/C
Uso 2 - CN	48/27	36/36	72/62	90/74	246/199
Uso 2 - PE	0/0	0/0	0/0	20/20	20/20
Uso 2 - PF	24/24	0/0	0/0	0/0	24/24
Uso 2 - CD	48/30	36/36	72/71	90/90	246/227
Uso 2 - Todos	120/81	72/72	144/133	200/184	536/470
JaBUTi - All-Nodes-ei	12/12	17/16	6/6	8/8	43/42
JaBUTi - All-Edges-ei	16/16	26/18	5/5	10/10	57/49
JaBUTi - All-Uses-ei	40/32	90/84	10/10	33/33	173/159
JaBUTi - All-Pot-Uses-ei	62/32	61/48	9/9	48/48	180/137
TOTAIS	370/254	338/310	318/296	499/467	1525/1327

Tabela 6.7: Resultados da avaliação do conjunto Ta do Uso 2 com os outros critérios.

Critérios/Programas	TriTyp	Ordena	Data	Fourballs	TOTAIS
	E/C	E/C	E/C	E/C	E/C
Uso 1	4/4	6/6	4/4	5/5	19/19
JaBUTi - All-Nodes-ei	12/12	17/16	6/6	8/8	43/42
JaBUTi - All-Edges-ei	16/16	26/18	5/5	10/10	57/49
JaBUTi - All-Uses-ei	40/33	90/84	10/10	33/33	173/160
JaBUTi - All-Pot-Uses-ei	50/34	61/48	9/9	48/48	180/139
TOTAIS	134/99	200/172	34/34	104/104	472/409

6.5 Análise dos Resultados

Os resultados apresentados na seção anterior são analisados e detalhados separadamente nas próximas sub-seções.

Com relação à comparação entre os critérios, dois fatores são considerados:

Tabela 6.8: Resultados da avaliação do conjunto Ta da JaBUTi com os outros critérios.

Critérios/Programas	**TriTyp**	**Ordena**	**Data**	**Fourballs**	**TOTAIS**
	E/C	E/C	E/C	E/C	E/C
Uso 1	4/4	6/6	4/4	5/5	19/19
Uso 2 - CN	48/27	36/36	72/62	90/74	246/199
Uso 2 - PE	0/0	0/0	0/0	20/20	20/20
Uso 2 - PF	24/24	0/0	0/0	0/0	24/24
Uso 2 - CD	48/30	36/36	72/71	90/90	246/227
Uso 2 - Todos	120/81	72/72	144/133	200/184	536/470
TOTAIS	244/166	150/150	292/270	405/373	1091/959

- Custo: número de dados de teste necessários para cobrir os elementos requeridos, considerando o número de elementos não executáveis;
- Dificuldade de satisfação: probabilidade de satisfazer um critério tendo satisfeito outro.

6.5.1 Uso 1

O Uso 1 trata da validação da classe conforme sua especificação. No experimento esse critério apresentou como uma das suas características a de necessitar menor esforço de teste para a cobertura dos elementos requeridos, ou seja menor custo. Ao todo foram necessários 18 dados de teste para cobrir todos os 19 elementos requeridos.

O fato do conjunto de dados de teste inicial Ti de cada classe do experimento possuir valores entre uma faixa limite dos valores válidos e inválidos aproximados de entrada para cada classe aparenta ter contribuído para esse menor esforço. A Tabela 6.2 apresenta esse menor esforço quando observado o Uso 1 da abordagem sobre os programas $TriTyp$, *Ordena* e *Data*. Enquanto no primeiro e no terceiro programas citados ocorreu a cobertura de 3 dos 4 elementos requeridos, para o segundo programa o resultado foi ainda melhor cobrindo 6 dos 6 elementos requeridos utilizando somente os dados do conjunto Ti. Com isso pode-se constatar que a dificuldade de satisfação é relativamente baixa.

Não foi objetivo do experimento avaliar a eficácia dos critérios, mas um defeito foi revelado com relação à classe *Fourballs* para qual foi definido um elemento requerido de pré-condição que na implementação da classe não havia sido tratado. Por esse motivo em algumas situações indevidas o método foi executado gerando um comportamento inadequado. Essa situação pode ser facilmente identificada nos resultados da ferramenta. No

mais, cada implementação se mostrou coerente com o especificado.

6.5.2 Uso 2

O Uso 2 da abordagem trabalha com a mutação da especificação. Esse critério apresentou no experimento diversos elementos requeridos alternativos (mutantes) a cada elemento requerido das classes avaliadas. Isso refletiu em um custo maior para alcançar a cobertura dos elementos requeridos elaborados.

Foram necessários 31 dados de entrada para cobertura de todos os elementos requeridos executáveis, e ainda, 51 (9,10%) elementos requeridos após avaliação foram considerados inconsistentes, e não puderam ser cobertos, sendo 32 (5,71%) deles oriundos de especificações que sofreram mutação do operador CN e 19 (3,39%) de especificações que sofreram mutação do operador CD.

Uma situação especial foi identificada no experimento com a classe *Ordena* na qual todos os elementos requeridos foram cobertos com o conjunto de dados de teste inicial Ti, conforme ilustrado na Tabela 6.2. Fora essa exceção, dados de teste mais elaborados tiveram de ser desenvolvidos para essa abordagem, aumentando a dificuldade de satisfação e esforço.

Nos resultados apresentados, diversas especificações modificadas foram totalmente cobertas. Porém em uma avaliação a cada uma delas não foi observado nenhuma que especificasse de melhor forma a especificação original. O problema (defeito) descrito na seção anterior quanto à pré-condição definida para a especificação da classe *Fourballs* e não implementada pôde ser revelado também com esse uso. Entretanto as especificações originais definidas não apresentaram defeitos em avaliação com as especificações mutantes elaboradas.

6.5.3 JaBUTi

Os resultados com a JaBUTi apresentam semelhanças com os outros dois critérios: com o Uso 1 com relação a uma boa cobertura com o conjunto de dados de entrada Ti e em possuir os mesmos dados de entrada adicionais aos programas conforme ilustra a

Tabela 6.4; e com o Uso 2 com relação à existência de elementos que não puderam ser cobertos, nesse caso elementos não executáveis.

Para cobertura dos elementos requeridos de todos os programas avaliados, foram utilizados 20 dados de entrada, sendo que 17 deles foram obtidos a partir do conjunto Ti e 3 foram os dados de entrada adicionais.

Os elementos não executáveis podem ser considerados os causadores de um maior custo de teste, no total foram 32 (7,06%), sendo 1 (0,22%) do critério All-Nodes-ei, 6 (1,32%) do critério All-Edges-ei, 5 (1,10%) do critério All-Uses-ei e 20 (4,42%) do critérios All-Pot-Uses sendo esse critério o que gerou o maior número de elementos requeridos também.

Quanto aos testes ainda, uma situação não contemplada pela JaBUTi é a de trabalhar com as pré-condições, conforme o que foi trabalhado no experimento com a classe "Fourballs", ou seja, considerar os valores que são válidos na entrada de um determinado método. Porém vale destacar que isso não é mesmo objetivo da JaBUTi, que não trabalha considerando a especificação da classe.

6.5.4 Comparação entre os critérios

Comparando os critérios quanto ao custo do teste, o Uso 1 necessita de um custo menor, com a elaboração de menos dados de teste e uma cobertura maior com o conjunto de dados de teste Ti. A JaBUTi apresenta um número de dados de teste um pouco maior que o Uso 1 (de 18 para 20), uma semelhante cobertura com o conjunto de dados de teste Ti e os mesmos dados de teste adicionais que o Uso 1 utiliza, porém o custo de teste da JaBUTi é maior pela presença de elementos não executáveis (32) (7,06%). Já o Uso 2 apresentou os maiores custos de testes, com o maior número de dados de teste (31) e o maior número de elementos inconsistentes (51) (9,10%). Estes resultados são semelhantes ao que acontece comparando teste de programas estrutural e baseado em defeitos com a aplicação da análise de mutantes tradicional.

Quanto à dificuldade de satisfação entre os critérios, conforme o experimento, pode-se considerar que os dados de teste adequados ao Uso 2 também são adequados ao Uso 1, ou seja, a satisfação do Uso 2 inclui a satisfação da abordagem Uso 1. Isso pode ser

comprovado comparando nos resultados das Tabelas 6.5 e 6.7. Os dados de entrada do Uso 2 também são adequados ao critério "All-Nodes-ei" da JaBUTi conforme resultados comparados nas Tabelas 6.5 e 6.7. E ainda, que os dados de entrada da JaBUTi são adequados para o Uso 1 da abordagem conforme resultados comparados entre as Tabelas 6.5 e 6.8. Já os demais critérios da JaBUTi e do Uso 2 são incomparáveis, eles não incluem o Uso 2 e nem o Uso 2 os inclui.

6.5.5 Considerações Finais

O presente capítulo descreveu detalhadamente o experimento desse trabalho, apresentando: os objetivos, os programas, a metodologia utilizada, os resultados e análise dos resultados.

Destaca-se na análise dos resultados o parecer referente ao custo comparativo para cobertura de cada uma dos critérios estudados, assim como a dificuldade de satisfação entre eles.

A partir dos resultados é perceptível como o Uso 1 da abordagem "Geração de dados de teste a partir da especificação" , tem um custo mais baixo de execução. A geração de testes com dados da especificação auxilia para que os testes sejam direcionados às funções que o programa precisa executar. O Uso 1 também pode ser considerado a de menor dificuldade de satisfação entre os critérios.

A geração de especificações modificadas a partir de operadores de mutação, gera diversos novos elementos requeridos, elementos esses, que exigem dados de teste mais bem elaborados para proporcionar a sua cobertura. O Uso 2 da abordagem apresentou exatamente esse aspecto, que soma no custo e na dificuldade de satisfação do teste, junto aos elementos inconsistentes que as especificações que sofreram mutação apresentaram. Se faz necessário a avaliação dos resultados de futuros experimentos, para considerar a possibilidade de se melhorar os operadores de mutação, quanto à geração de elementos inconsistentes e para descrever outros tipos de defeitos. Nesse experimento o fator eficácia (número de defeitos revelados), não foi avaliado.

A JaBUTi apesar de semelhantes números de dados de teste necessários para cobertura

dos seus critérios com relação ao Uso 1, apresentou diversos elementos não executáveis a mais, que ampliaram o custo e cobertura do teste. É válido lembrar que existem diferenças entre os objetivos dos critérios da JaBUTi e dos critérios propostos pelos usos 1 e 2 da abordagem, o experimento levou em consideração somente o custo e a dificuldade de satisfação do teste. Isso ficou bem claro pois um defeito que era uma pré-condição ausente na implementação foi revelado pelos usos da abordagem proposta nesse trabalho. Pode-se concluir que eles devem ser utilizados de maneira complementar.

CAPÍTULO 7

CONCLUSÃO

O presente trabalho contribuiu para a atividade de teste de programas OO, com base em teste de mutação de contratos de especificação. O trabalho foi desenvolvido levando em conta uma abordagem estudada com dois usos bem definidos: o primeiro considerando a geração de dados de teste a partir da especificação para o teste de programa OO (Uso 1), e o segundo considerando a validação da especificação e da implementação por meio da análise de especificações que sofreram mutação (Uso 2).

As espeficicações adotadas para a aplicação das abordagens foram baseadas em um subconjunto da OCL, apresentado no Capítulo 4.

A POA foi uma das tecnologias utilizadas na instrumentação dos testes. Isso possibilitou a instrumentação das assertivas necessárias para o controle de pré-condições e pós-condições, sem a necessidade de alterar o código fonte. Além de possibilitar um custo menor visto possibilitar a execução de diversos aspectos em uma única execução da classe em teste.

Muitos foram os desafios enfrentados, entre eles destaca se o desenvolvimento de uma ferramenta para apoiar o uso prático da abordagem. A ferramenta MuSA foi desenvolvida para instrumentar os dois usos propostos na abordagem. Uma dessas limitações é partir o teste somente de programas desenvolvidos em Java.

O experimento realizado no trabalho visou à aplicação e validação dos usos da abordagem estudada com base na ferramenta MuSA. Conforme a metodologia utilizada no experimento realizado, os usos para a abordagem proposta puderam ser vistos como critérios de teste, pois satisfazer as pré e pós-condições foram avaliados como elementos requeridos em cada uso. Os critérios correspondentes a cada uso da abordagem foram comparados com os critérios All-Nodes-ei, All-Edges-ei, All-Uses-ei e All-Pot-Uses-ei da JaBUTi, com ênfase ao custo do teste e a dificuldade de satisfação.

Através do experimento pode-se constatar que o Uso 1 da abordagem tem um custo de teste e dificuldade de satisfação menor que os critérios avaliados da JaBUTi, e já o Uso 2 da abordagem tem um custo e uma dificuldade de satisfação maior que o Uso 1 da abordagem e os critérios avaliados da JaBUTi. Estes resultados são semelhantes ao que acontece comparando teste de programas estrutural e baseado em defeitos com a aplicação da análise de mutantes tradicional.

O experimento ainda apresentou como resultado para o Uso 2 da abordagem um número considerável de elementos inconsistentes (9,10%) nas especificações que sofreram mutação com os operadores CN e CD, percentagem similar aos elementos não executáveis requeridos pelos critérios da JaBUTi (7,06%).

Para finalizar, com base em todo o estudo realizado e os resultados obtidos pelo experimento, é visto que existem muitos trabalhos ainda a serem desenvolvidos na área de teste de especificação. Esse trabalho contribui nesse contexto, considerando a mutação de especificações em OCL para teste da especificação original e do programa.

Mesmo não tendo sido objetivo do experimento avaliar a eficácia dos usos da abordagem com relação ao número de defeitos revelados, foi observado que a implementação de um dos programas testados, não havia realizado um teste antes de chamar um método instrumentado, ou seja, a pré-condição estava ausente. Isso comprovou na prática que a utilização dos usos da abordagem proposta nesse trabalho auxilia a revelar caminhos ausentes.

7.1 Trabalhos Futuros

O trabalho realizado por resultar em uma nova ferramenta, com abordagem nova, possui algumas melhorias futuras que podem ser trabalhadas. Abaixo algumas sugestões de trabalhos futuros são apresentadas.

A ferramenta MuSA foi desenvolvida para a aplicação dos dois usos da abordagem somente em classes da linguagem Java. A mesma abordagem poderia ser aplicada também para outras linguagens de desenvolvimento em OO que permitissem o desenvolvimento utilizando POA.

Ainda sobre a ferramenta MuSA, a mesma pode receber diversas melhorias quanto à operacionalidade, novos processos e relatórios que sejam de interesse dos usuários. Um processo interessante que poderia ser desenvolvido seria o da interpretação da linguagem OCL pela própria ferramenta, sem a necessidade de informar o XML referente a especificação OCL.

O conjunto de operadores de mutação desenvolvido contém algumas limitações com relação ao domínio que atuam. A partir de novos experimentos, melhorias nesse conjunto de operadores e até a introdução de novos devem ser realizadas.

Os operadores de mutação CN e CD no experimento realizado, apresentam um número significativo de elementos requeridos inconsistentes. O acompanhamento desses operadores em outros experimentos faz-se importante, para prever possíveis melhorias, utilizando até alguma heurística que condicione quando do uso desses operadores para reduzir o número de elementos inconsistentes gerados.

Outros experimentos devem ser conduzidos para avaliar o fator eficácia e os usos da abordagem para outros programas mais complexos.

BIBLIOGRAFIA

[ABD+79] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. *Relatório Técnico GIT-ICS-79/08*, 1979.

[ADH+89] H. Agrawal, R. A. DeMillo, R. Hataway, Wm. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford. Design of mutant operators for C programming language. March 1989.

[AS05] B. K. Aichernig and P. A. P. Salas. Test Case Generation by OCL Mutation and Constraint Solving. In *QSIC '05: Proceedings of the Fifth International Conference on Quality Software*, pages 64–71, Washington, DC, USA, 2005. IEEE Computer Society.

[BBH02] M. Benattou, J. Bruel, and N. Hameurlain. Generating Test Data from OCL Specification, 2002. Disponível em: citeseer.ist.psu.edu/600470.html. Acessado em Março de 2008.

[BG02] O. Beckman and B. Grupta. Developing test cases from use cases for web applications. In *International Conference on Pratical Software Testing Techniques - PSTT'2002*, New Orleans, 2002.

[Bin99] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*, volume 1. Addison-Wesley LongMan, Inc., 1999.

[BLM+07] D. Barbosa, H. Lima, P. Machado, J. Figueiredo, M. Juca, and W. Andrade. Automating functional testing of componentes from UML specifications. In *Internatioal Journal of Software Engineering and Knowledge Engineering*, pages 339–358, June 2007.

[CCJ03] M. L. Chaim, A. Carniello, and M. Jino. Teste baseado em casos de uso. *Boletim de Pesquisa e Desenvolvimento - Embrapa Informática Agropecuária*, Dezembro 2003.

[CR83] L.A. Clarke and D.J. Richardson. The Application of Error-Sensitive Testing strategies to debugging. In *Symposium of High-Level Debugging*, pages 45–52. ACM SIGSOFT/SIGPLAN, May 1983.

[DBL05] W. J. Dzidek, L. C. Briand, and Y. Labiche. Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In Jean-Michel Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *Lecture Notes in Computer Science*, pages 10–19. Springer, 2005.

[Dio04] F. Diotalevi. Contract enforcement with AOP. Apply Design by Contract to Java software development with AspectJ. 2004. Disponível em: "http://www.ibm.com/developerworks/library/j-ceaop/". Acessado em Maio de 2009.

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. volume 11(4), pages 34–43. IEEE Computer, April 1978.

[DMJ07] M. E. Delamaro, J. C. Maldonado, and M. Jino. *Introdução ao Teste de Software*. Editora Campus - Elsevier, 2007.

[DO91] R. A. DeMillo and A. J. Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, 1991.

[dPPF03] W. de P. Paula Filho. *Engenharia de Software Fundamentos, Métodos e Padrões*. LTC - Livros Técnicos e Científicos Editora S.A., 2003.

[Fou08] Eclipse Foundation. Chapter 2. The Aspectj Language, 2008. Disponível em: "http://www.eclipse.org/aspectj/doc/released/progguide/language.html". Acessado em Março de 2008.

[GBR03] M. Gogolla, J. Bohling, and M. Richters. Validation of UML and OCL Models by Automatic Snapshot Generation. In *In Proceedings of the 6th*

Int. Conf. Unified Modeling Language (UML 2003, pages 265–279. Springer, 2003.

[GG75] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software*, pages 493–510, New York, NY, USA, 1975. ACM Press.

[Har00] M. J. Harrold. Testing: A roadmap. In *In The Future of Software Engineering*, pages 61–72. ACM Press, 2000.

[Heu01] J. Heumann. Is a use case a test case? In *International Conference on Pratical Software Testing Techniques - PSTT'2001*. St. Paul: Software Dimension, 2001.

[HMF92] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental testing of object-oriented class structures. In *14th International Conference on Software Engineering*, pages 68–80. IEEE Computer Society Press, May 1992.

[How87] W. E. Howden. *Software Engineering and Technology: Functional Program Testing and Analysis*. MacGrall-Hill Book Co, 1987.

[HR94] M. J. Harrold and G. Rothermel. Performing dataflow testing on classes. In *Proceedings of the SIGSOFT'94 Symposium on the Foundations of Software Engineering*, pages 154–163, New Orleans, NO, USA, 1994. ACM Press.

[iee90] *IEEE 90: IEEE Standard Glossary of Software Engineering Terminology*. 1990.

[JHS+05] Y. Jiang, S. Hou, J. Shan, L. Zhang, and B. Xie. Contract-based mutation for testing components. In *ICSMapos '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 483–492, Budapest, 2005.

[LRM07] O. A. L. Lemos, R. Ré, and P. C. Masiero. Teste de Aspectos. In *M. C. Delamaro, J. C. Maldonado e M. Jino, Introdução ao Teste de Software*, pages 47–76. Ed. Campus-Elsevier, Setembro 2007.

[Mal91] J. C. Maldonado. Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software. PhD Thesis, DCA/FEE/UNICAMP, Julho 1991.

[MB89] T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, Dezembro 1989.

[MBV+] J. Maldonado, E. Barbosa, A. Vincenzi, M. Delamaro, S. Souza, and M. Jino. Introdução ao Teste de Software. Notas Didáticas do Instituto de Ciências Matemáticas e de Computação - ICMC/USP.

[MS01] J. D. McGregor and D. A. Sykes. *A Pratical Guide to Testing Object-Oriented Software*. Addison-Wesley, 2001.

[MW93] A. P. Mathur and W. E. Wong. Evaluation of the cost of alternative mutation strategies. pages 320–335. VII Simpósio Brasileiro de Engenharia de Software, 1993.

[Mye04] G. J. Myers. *The Art of Software Testing; Revised and updated by Tom Badgett and Todd Thomas, with Corey Sandler. 2nd ed.* John Wiley & Sons, Inc., 2004.

[MZ05] H. Mei and L. Zhang. A Framework for Testing Web Services and Its Supporting Tool. In *SOSE '05: Proceedings of the IEEE International Workshop*, pages 207–214, Washington, DC, USA, 2005. IEEE Computer Society.

[Nta84] S. C. Ntafos. On required element testing. volume SE-10(6), pages 795–803. IEEE Transactions on Software Engineering, November 1984.

[OLR+96] A. J. Offtutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination o sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5, n. 2:99–118, 1996.

[OMG08] The Object Management Group OMG. Object Constraint Language, 2008. Disponível em: "http://www.omg.org/technology/documents/formal/ocl.htm". Acessado em Abril de 2008.

[ORZ93] A. J. Offtutt, G. Rothermel, and C. Zapf. An Experimental Evaluation of Selective Mutation. *IEEE Computer Society Press*, pages 100–107, 1993.

[PCM96] A. Pasquini, A. N. Crespo, and P. Matrella. Sensitivity of reliability growth models to operation profile erros vs testing accuracy. pages 45:531–540. IEEE Transactions on Reliability, 1996.

[Pre06] R. S. Pressman. *Engenharia de Software*. McGraw-Hill, 2006.

[RdSSMM05] A. D. Rocha, A. da S. Simão, J. C. Maldonado, and P. C. Masiero. Uma ferramenta baseada em aspectos para o teste funcional de programas Java - ICMC/USP. XIX Simpósio Brasileiro de Engenharia de Software (SBES2005) - Universidade Federal de Uberlândia, 2005.

[RG03] M. Richters and M. Gogolla. Aspect-Oriented Monitoring of UML and OCL Constraints. In *In AOSD Modeling With UML Workshop, 6th International Conference on the Unified Modeling Language (UML*, 2003.

[RM98] A. C. A. Rosa and E. Martins. Using a Reflective Architeture to Validade Object-Oriented Applications by Fault Injection. pages 76–80. Proc. of the Workshop on Reflective Programming in C++ and Java, October 1998.

[RW85] S. Rapps and E. Weyuker. Selecting software test data using data flow information. volume SE-11(4), pages 367–375. IEEE Transactions on Software Engineering, Abril 1985.

[SFB+07] S.R.S. Souza, S.C.P.F. Fabbri, E.F. Barbosa, M.C. Chaim., A.M.R. Vincenzi, M.E. Delamaro, M. Jino, and J.C. Maldonado. Estudos Teóricos e Experimentais. In *M. C. Delamaro, J. C. Maldonado e M. Jino, Introdução ao Teste de Software*, pages 252–268. Ed. Campus-Elsevier, Setembro 2007.

[VDDM07] A. M. R. Vincenzi, A. L. S. Domingues, M. E. Delamaro, and J. C. Maldonado. Teste Orientado a Objetos e de Componentes. In *M. C. Delamaro, J. C. Maldonado e M. Jino, Introdução ao Teste de Software*, pages 119–174. Ed. Campus-Elsevier, Setembro 2007.

[Vin04] A. M. R. Vincenzi. Orientação a Objeto: Definição, Implementação e Análise de Recursos de Teste e Validação. 2004. Disponível em: http://www.teses.usp.br/teses/disponiveis/55/55134/tde-17082004-122037/publico/tese.pdf. Acessado em Março de 2008.

[VJ03] R. L. Van and T. Jensen. UML-Casting: Test synthesis from UML models using constraint, 2003. Disponível em: citeseer.ist.psu.edu/665413.html. Acessado em Março de 2008.

[VMWD05] A. M. R. Vincenzi, J. C. Maldonado, W. E. Wong, and M. E. Delamaro. Coverage testing of Java programs and components. *Jornal of Science of Computer Programming*, 56(1-2):211–230, April 2005.

[WM95] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. volume 31, n. 3, pages 185–196. Journal o Software and Systems, 1995.

[WMM94] W.E. Wong, A.P. Mathur, and J.C. Maldonado. Mutation Versus All-uses: An Empirical Evaluation of Cost, Strength and Effectiveness. In *Software Quality and Productivity - Theory, Practice, Education and Training*. Hong Kong, December 1994.

APÊNDICE A

ASPECTJ

AspectJ é uma extensão de Java para apoiar a POA. Por ser uma extensão de Java, AspectJ contém todas as construções dessa linguagem, e portando é contruída sobre o paradigma OO [LRM07].

Com AspectJ, além da possibilidade de utilizar todos os recursos oriundos da linguagem Java, é possível utilizar algumas palavras reservadas como *aspect*. Aspectos são módulos que podem alterar a estrutura estática ou dinâmica de um programa, combinam pontos de junção, adentos anteriores, posteriores e de contorno; declarações inter-tipo para introduzir atributos, métodos e heranças de outras classes para os aspectos [LRM07].

Os pontos de junção podem ser definidos como pontos de execução de um programa onde os aspectos são chamados. Os pontos de junção podem efetuar com a seguintes operações: chamada de métodos, execução de métodos, chamada de construtores, execução de inicializadores, execução de construtores, execução de inicialização estática, pré-inicialização de objetos, inicialização de objetos, referência a campos e execução de tratamento de exceções.

Os *pointcuts* são responsáveis pela seleção dos pontos de junção, eles detectam em que ponto do programa os aspectos devem interceptar. Podem ser declarados como públicos, privados e até abstratos (quando o aspecto também for abstrato). Podem ser aninhados pelos operadores de adição, de condição ou de negação.

Para a definição de um *pointcut* alguns designadores são necessários, como: *Call(Signature), Execution(Signature), Get(Signature), Set(Signature), This(Type pattern), Target(Type pattern), Args(Type pattern) e Within(Type pattern)* que são descritos em [Fou08].

O comportamento multidimensional de um aspecto é implementado pode um *advice* (adendo). Um *advice* é um trecho de código que é executado a cada ponto de junção descrito por um *pointcut*. As três formas de *advice* são as seguintes:

- *before:* é um *advice* que expressará um comportamento antes do ponto de junção;
- *after:* é um *advice* que expressará um comportamento depois do ponto de junção, o *advice after* pode ser definido para ser executado após o retorno normal do ponto de junção, ou após o retorno com uma excessão do ponto de junção;
- *around:* permite que o código do ponto de junção original seja executado dentro do *advice* por meio da chamada da palavra reservada *proceed* seguida dos argumentos da chamada original ao ponto de junção.

APÊNDICE B

ESQUEMA XML UTILIZADO PELA FERRAMENTA MUSA

A ferramenta MuSA precisa de determinadas informações para instrumentar os aspectos que formalizam os testes de uma determinada classe. O esquema XML representado abaixo formaliza as informações necessárias na especificação.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://esquema.org/"
        xmlns="">
  <xsd:element name="root">
     <xsd:complexType>
        <xsd:sequence>
        <xsd:element name="classe" maxOccurs="unbounded">
     <xsd:complexType>
                <xsd:sequence>
    <xsd:element name="atributo" maxOccurs="unbounded" minOccurs="0">
<xsd:complexType>
         <xsd:attribute name="id" type="xsd:string" use="required" />
         <xsd:attribute name="tipo" type="xsd:string" use="required" />
         </xsd:complexType>
        </xsd:element>
    <xsd:element name="metodo" maxOccurs="unbounded" minOccurs="0">
<xsd:complexType>
<xsd:sequence>
     <xsd:element name="parametro" maxOccurs="unbounded" minOccurs="0">
     <xsd:complexType>
```

```
          <xsd:attribute name="id" type="xsd:string" use="required" />
          <xsd:attribute name="tipo" type="xsd:string" use="required" />
          </xsd:complexType>
    </xsd:element>
</xsd:sequence>
     <xsd:attribute name="id" type="xsd:int" use="required" />
     <xsd:attribute name="nome" type="xsd:string" use="required" />
    <xsd:attribute name="retorno" type="xsd:string"/>
</xsd:complexType>
     </xsd:element>
<xsd:element name="precondicao" maxOccurs="unbounded" minOccurs="0">
<xsd:complexType>
<xsd:sequence>
 <xsd:element name="interface" maxOccurs="1" minOccurs="1">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="dado" maxOccurs="unbounded" minOccurs="0">
        <xsd:complexType>
          <xsd:attribute name="id" type="xsd:string" use="required" />
           <xsd:attribute name="tipo" type="xsd:string" use="required" />
           <xsd:attribute name="modo" type="xsd:string" use="required" />
           <xsd:attribute name="origem" type="xsd:string" use="required" />
         </xsd:complexType>
       </xsd:element>
     </xsd:sequence>
   </xsd:complexType>
 </xsd:element>
 <xsd:element name="validacao" maxOccurs="unbounded">
  <xsd:complexType>
```

```
   <xsd:sequence></xsd:sequence>
    <xsd:attribute name="id" type="xsd:string"/>
    <xsd:attribute name="comando" type="xsd:string"/>
    <xsd:attribute name="comandoMutado" type="xsd:string"/>
    <xsd:attribute name="operadorMutante" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>
<xsd:element name="restricao" maxOccurs="unbounded">
 <xsd:complexType>
  <xsd:sequence></xsd:sequence>
  <xsd:attribute name="id" type="xsd:string"/>
  <xsd:attribute name="comando" type="xsd:string"/>
 </xsd:complexType>
</xsd:element>
<xsd:element name="escopo" maxOccurs="1" minOccurs="1">
 <xsd:complexType>
   <xsd:sequence>
     <xsd:element name="chamada" maxOccurs="unbounded" minOccurs="0">
       <xsd:complexType>
         <xsd:attribute name="expressaoAspectJ" type="xsd:string" use="optional"/>
          <xsd:attribute name="metodoId" type="xsd:int"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="definicao" minOccurs="0">
 <xsd:complexType>
   <xsd:sequence>
```

```
      <xsd:element name="atributo" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence/>
           <xsd:attribute name="id" type="xsd:string"/>
           <xsd:attribute name="valor" type="xsd:string"/>
         </xsd:complexType>
       </xsd:element>
     </xsd:sequence>
   </xsd:complexType>
 </xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:string" use="required" />
</xsd:complexType>
</xsd:element>
<xsd:element name="poscondicao" maxOccurs="unbounded" minOccurs="0">
<xsd:complexType>
<xsd:sequence>
 <xsd:element name="interface" maxOccurs="1" minOccurs="1">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="dado" maxOccurs="unbounded" minOccurs="0">
        <xsd:complexType>
          <xsd:attribute name="id" type="xsd:string" use="required" />
           <xsd:attribute name="tipo" type="xsd:string" use="required" />
           <xsd:attribute name="modo" type="xsd:string" use="required" />
           <xsd:attribute name="origem" type="xsd:string" use="required" />
         </xsd:complexType>
       </xsd:element>
     </xsd:sequence>
```

```
  </xsd:complexType>
</xsd:element>
<xsd:element name="validacao" maxOccurs="unbounded" minOccurs="1">
 <xsd:complexType>
  <xsd:sequence></xsd:sequence>
  <xsd:attribute name="id" type="xsd:string"/>
  <xsd:attribute name="comando" type="xsd:string"/>
  <xsd:attribute name="comandoMutado" type="xsd:string"/>
  <xsd:attribute name="operadorMutante" type="xsd:string"/>
 </xsd:complexType>
</xsd:element>
<xsd:element name="restricao" maxOccurs="unbounded">
 <xsd:complexType>
  <xsd:sequence></xsd:sequence>
  <xsd:attribute name="id" type="xsd:string"/>
  <xsd:attribute name="comando" type="xsd:string"/>
 </xsd:complexType>
</xsd:element>
<xsd:element name="escopo" maxOccurs="1" minOccurs="1">
 <xsd:complexType>
   <xsd:sequence>
     <xsd:element name="chamada" maxOccurs="unbounded" minOccurs="0">
       <xsd:complexType>
         <xsd:attribute name="metodoId" type="xsd:int" use="optional"/>
          <xsd:attribute name="expressaoAspectJ" type="xsd:string" use="optional"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
```

```
 </xsd:element>
 <xsd:element name="definicao" minOccurs="0">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="atributo" maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:sequence/>
      <xsd:attribute name="id" type="xsd:string"/>
      <xsd:attribute name="valor" type="xsd:string"/>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:string" use="required" />
</xsd:complexType>
</xsd:element>
</xsd:sequence>
      <xsd:attribute name="id" type="xsd:string"/>
     </xsd:complexType>
</xsd:element>
</xsd:sequence>
   </xsd:complexType>
</xsd:element>
</xsd:schema>
```

O esquema XML do modelo de especificação têm vários nós e atributos XML definidos que possuem os seguintes significados para instrumentação do aspecto:

- *root*: Nó principal e único por arquivo de especificação.

- *classe*: Nó especificando o início de uma nova classe, possui o atributo *id* cujo conteúdo deve corresponder ao nome da classe. Vários nós *classe* podem ser especificado no mesmo arquivo.

- *atributo*: Nó especificando um atributo da classe que está sendo especificada, possui o atributo *id* que deve conter o nome do atributo da classe e o atributo *tipo* deve conter o tipo de dado em Java do atributo (ex: String, int, float, boolean, etc).

- *metodo*: Nó especificando um método da classe, possui os atributos: *id* que deve conter um identificador único ao método, o atributo *nome* que deve conter o nome do método e o atributo *retorno* que deve conter o tipo do retorno esperado ao método.

- *parametro*: Nó especificando um parâmetro de entrada de um método, pode ser especificado várias vezes dentro de um método. Possui os seguintes atributos: *id* que deve conter o nome do parâmetro de entrada e *tipo* que corresponde do tipo de dado Java correspondente ao parâmetro.

- *precondicao*: Nó especificando uma pré-condição que pode ser aplicada a vários métodos da classe. Cada classe pode possuir vários nós de pré-condições cadastrados. Possui o atributo *id* que deve conter um identificador único, que distingüe a pré-condição na classe.

- *interface*: Nó especificando o agrupamento das entradas esperadas na chamada de uma pré-condição.

- *dado*: Nó especificando uma entrada de uma pré-condição, possui os atributos: *id* que é o nome definido ao dado de entrada, *tipo* que se refere ao tipo esperado do dado de entrada, *modo* que ilustra como deve ser o comportamento que o dado deverá ter na implementação do aspecto, podendo ser atribuido como "atributo", "objeto"ou "preelemento"; e *origem* que simboliza de que pacote Java deve ser importado o dado, sendo que quando tratar de um tipo do dado da biblioteca *java.lang* ou da

própria classe que está sendo especificada, pode ser informado um ".", caso contrário deve ser informado o caminho completo (ex: *java.io*).

- *validacao*: Nó especificando o agrupamento de todas as validações que a pré-condição deve exercer. Possui os atributos: *id* identificador da validação, *comando* cujo conteúdo é ó código em java que representa a validação, *comandoMutado* atributo alimentado pela ferramenta quando executa-da com a opção de mutação de especificação, e *operadorMutante* também alimentado pela ferramenta em sua execução identificando que operador mutante foi utulizado na mutação.

- *restricao*: Nó especificando as restrições contidas na classe. As restrições devem ser formadas pelo agrupamento de uma ou mais validações definidas nos Nós de "validação". Tem os atributos: *id* com o identificador da restrição e *comando* com a especificação de quais validações fazem parte dessa restrição.

- *escopo*: Nó especificando o agrupamento de todos os possíveis escopos de utilização da pré-condição. Esse agrupamento é importante, para obrigar uma mesma pré-condição ser usada em mais de um método.

- *chamada*: Nó especificando um escopo válido para a chamada da pré-condição. Possui dois atributos que não podem co-existir em uma mesma chamada: *metodoid* que deve conter o valor de um atributo *id* de um nó *metodo* préviamente definido, e *expressaoAspectJ* que deve conter uma expressão AspectJ válida. A utilização da expressão é de grande validade quando da intenção de atribuir a pré-condição a um conjunto de métodos.

- *poscondicao*: Nó especificando uma pós-condição que pode ser aplicada a vários métodos da classe. Cada classe pode possuir vários nós de pós-condições cadastrados. Possui o atributo *id* que deve conter um identificador único, que distingüe a pós-condição na classe. Os demais nós são identicos funcionalmente aos nós definidos para o nó *precondicao*.

O esquema XML definido para a especificação foi definido com base em todas as necessidades encontradas para se instrumentar um teste sobre uma determinada classe seguindo as abordagens desse trabalho.

APÊNDICE C

TELAS DE RESULTADO DA FERRAMENTA MUSA

As Figuras C.1, C.2 e C.3 ilustram algumas das telas de resultados fornecidos pela ferramenta MuSA.

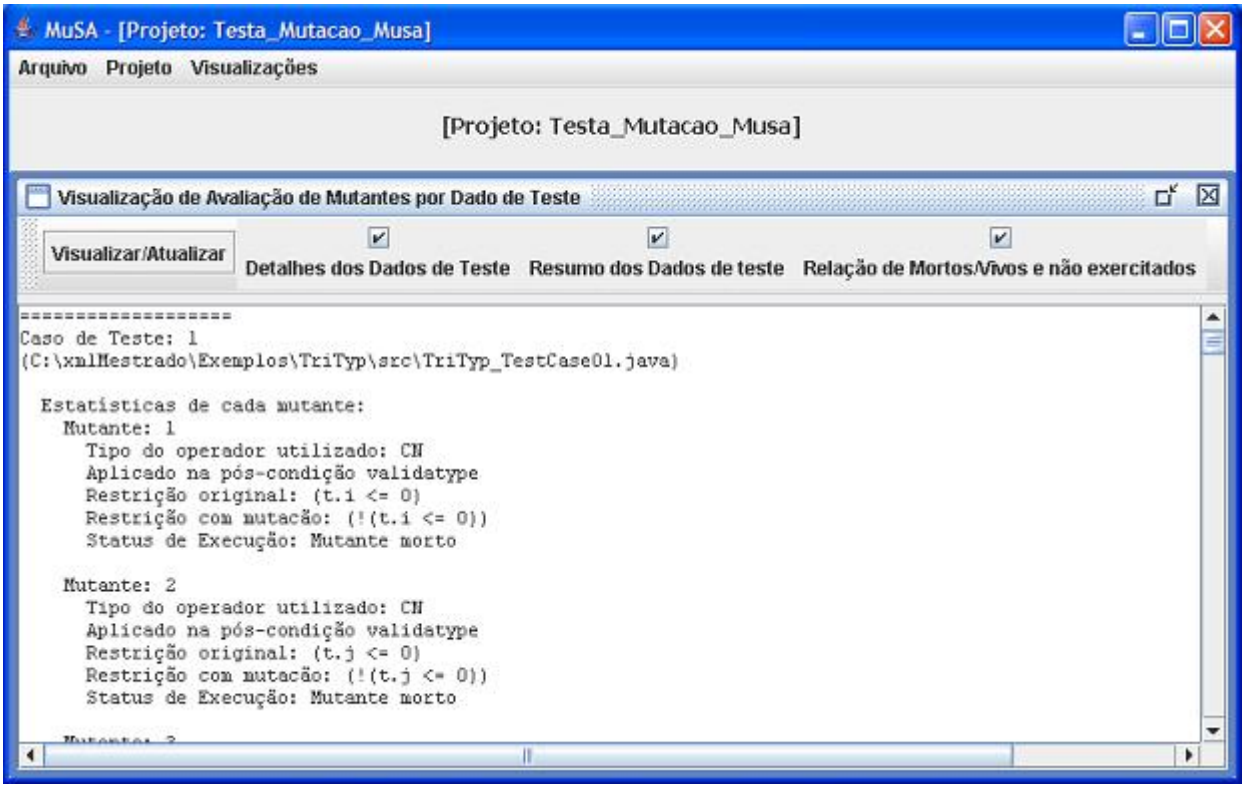


Figura C.1: MuSA - visualização de avaliação de mutantes por dado de teste - detalhes dos dados de teste.

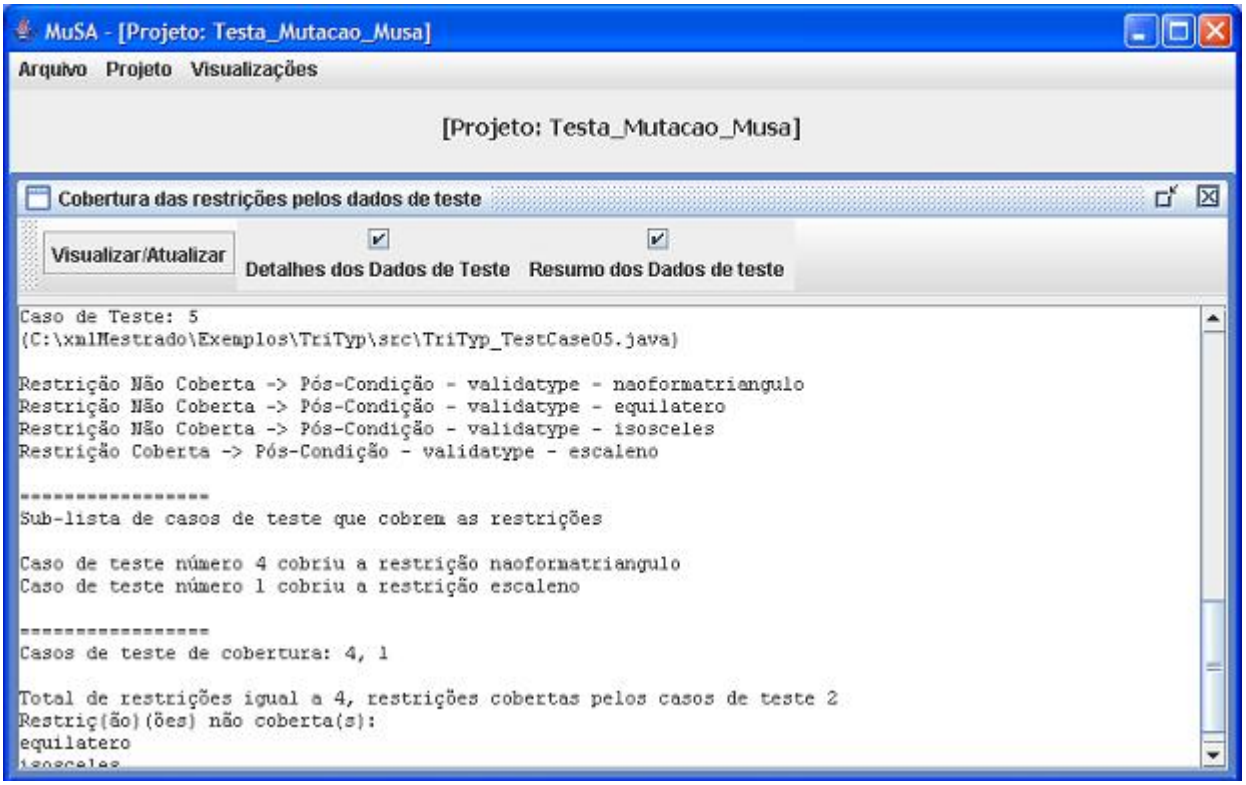


Figura C.2: MuSA - Cobertura das restrições por dado de teste.

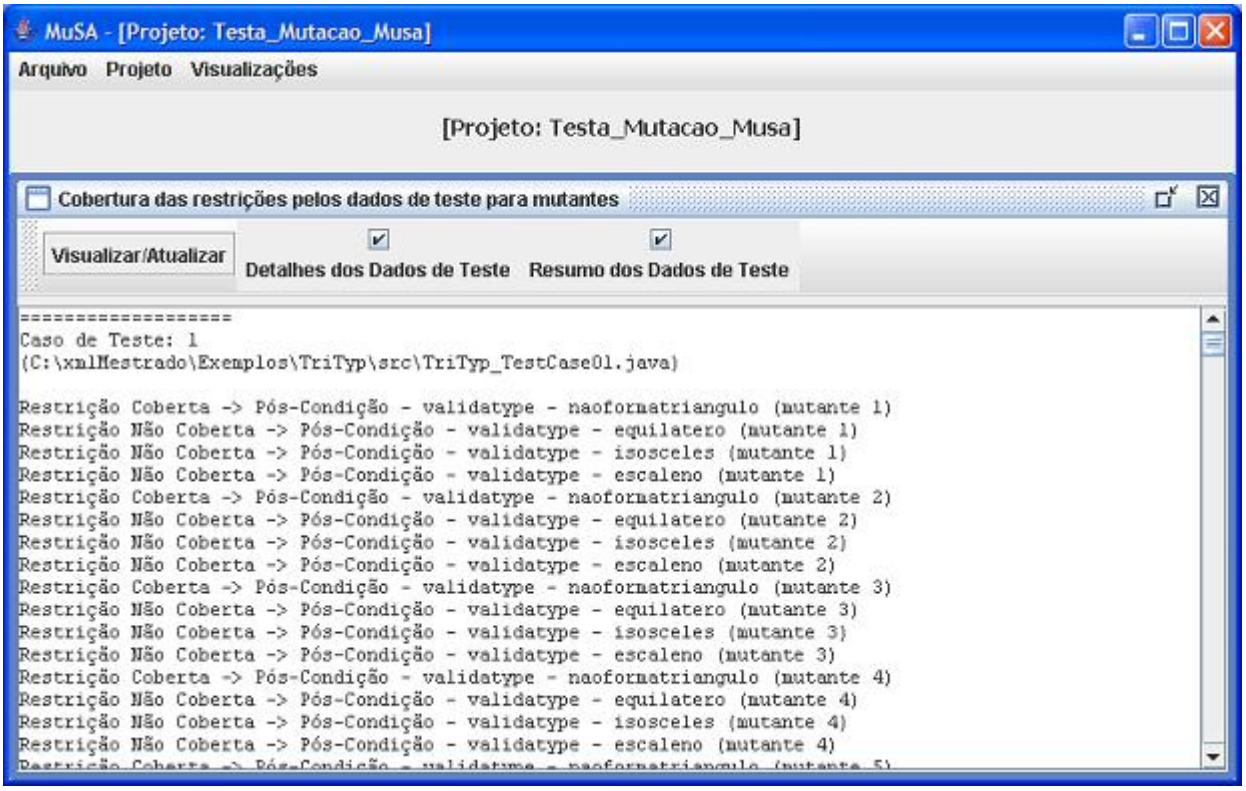


Figura C.3: MuSA - Cobertura das restrições das especificações modificadas por dado de teste.